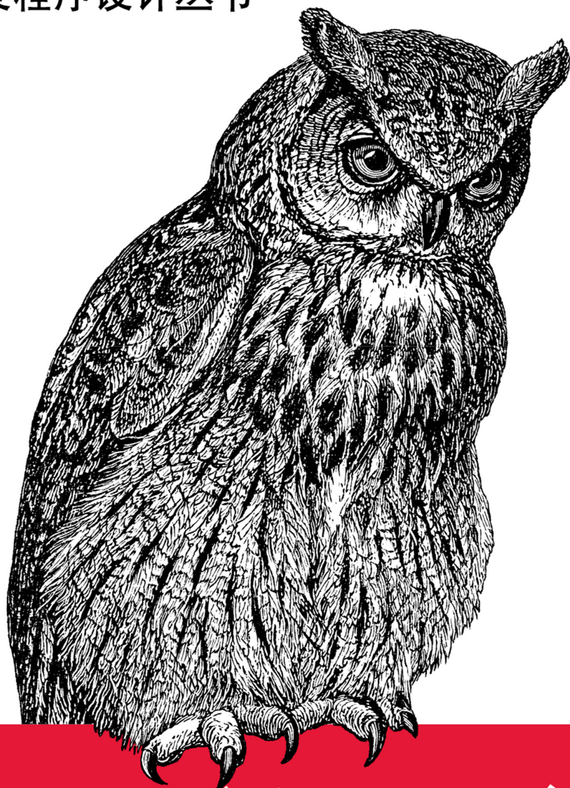
O'REILLY®



图灵程序设计丛书



机器学习实践

测试驱动的开发方法

Thoughtful Machine Learning

[美] Matthew Kirk 著

段菲 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

段菲

原三星电子（中国）研究院资深研究员，研究方向为机器学习和计算机视觉。



图灵程序设计丛书

机器学习实践
测试驱动的开发方法

Thoughtful Machine Learning
A Test-Driven Approach

[美] Matthew Kirk 著

段菲 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社

北　　京

图书在版编目（C I P）数据

机器学习实践 ： 测试驱动的开发方法 / (美) 柯克
(Kirk,M.) 著 ; 段菲译. -- 北京 : 人民邮电出版社,
2015.8
（图灵程序设计丛书）
ISBN 978-7-115-39618-1

Ⅰ. ①机… Ⅱ. ①柯… ②段… Ⅲ. ①机器学习—研
究 Ⅳ. ①TP181

中国版本图书馆CIP数据核字(2015)第148307号

内 容 提 要

本书主要介绍如何将测试驱动开发运用于机器学习算法。每一章都通过示例介绍了机器学习技术能够解决的有关数据的具体问题，以及求解问题和处理数据的方法。具体涵盖了测试驱动的机器学习、机器学习概述、*K* 近邻分类、朴素贝叶斯分类、隐马尔可夫模型、支持向量机、神经网络、聚类、核岭回归、模型改进与数据提取等内容。通过学习本书，你将能够利用机器学习技术解决涉及数据的现实问题。

本书适合开发人员、CTO 和商业分析师阅读。

◆ 著　　[美] Matthew Kirk
译　　段 菲
责任编辑　岳新欣
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：12.75
字数：302千字　2015年 8 月第 1 版
印数：1 – 3 500册　2015年 8 月北京第 1次印刷
著作权合同登记号　图字：01-2015-3679号

定价：49.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言 xi

第 1 章　测试驱动的机器学习 1

1.1　TDD 的历史 2

1.2　TDD 与科学方法 2

1.2.1　TDD 可构建有效的逻辑命题 3

1.2.2　TDD 要求你将假设以文字或代码的形式记录下来 5

1.2.3　TDD 和科学方法的闭环反馈机制 5

1.3　机器学习中的风险 5

1.3.1　数据的不稳定性 6

1.3.2　欠拟合 6

1.3.3　过拟合 7

1.3.4　未来的不可预测性 8

1.4　为降低风险应采用的测试 8

1.4.1　利用接缝测试减少数据中的不稳定因素 8

1.4.2　通过交叉验证检验拟合效果 9

1.4.3　通过测试训练速度降低过拟合风险 10

1.4.4　检测未来的精度和查全率漂移情况 11

1.5　小结 11

第 2 章　机器学习概述 13

2.1　什么是机器学习 13

2.1.1　有监督学习 13

2.1.2　无监督学习 14

2.1.3　强化学习 15

2.2 机器学习可完成的任务 15
2.3 本书采用的数学符号 16
2.4 小结 16

第 3 章 *K* 近邻分类 17

3.1 *K* 近邻分类的历史 18
3.2 基于邻居的居住幸福度 18
3.3 如何选择 *K* 21
3.3.1 猜测 *K* 的值 21
3.3.2 选择 *K* 的启发式策略 21
3.3.3 *K* 的选择算法 24
3.4 何谓“近” 24
3.4.1 Minkowski 距离 25
3.4.2 Mahalanobis 距离 26
3.5 各类别的确定 27
3.6 利用 KNN 算法和 OpenCV 实现胡须和眼镜的检测 29
3.6.1 类图 29
3.6.2 从原始图像到人脸图像 30
3.6.3 `Face` 类 33
3.6.4 `Neighborhood` 类 36
3.7 小结 43

第 4 章 朴素贝叶斯分类 45

4.1 利用贝叶斯定理找出欺诈性订单 45
4.1.1 条件概率 46
4.1.2 逆条件概率 47
4.2 朴素贝叶斯分类器 48
4.2.1 链式法则 48
4.2.2 贝叶斯推理中的朴素性 49
4.2.3 伪计数 50
4.3 垃圾邮件过滤器 51
4.3.1 类图 51
4.3.2 数据源 52
4.3.3 `Email` 类 52
4.3.4 符号化与上下文 55
4.3.5 `SpamTrainer` 类 56
4.3.6 通过交叉验证将错误率最小化 63
4.4 小结 66

第 5 章　隐马尔可夫模型 67

5.1　利用状态机跟踪用户行为 67

5.1.1　隐含状态的输出和观测 69

5.1.2　利用马尔可夫假设简化问题 70

5.1.3　利用马尔可夫链而非有限状态机 71

5.1.4　隐马尔可夫模型 71

5.2　评估：前向－后向算法 72

5.3　利用维特比算法求解解码问题 75

5.4　学习问题 76

5.5　利用布朗语料库进行词性标注 76

5.5.1　词性标注器的首要问题：`CorpusParser` 77

5.5.2　编写词性标注器 79

5.5.3　通过交叉验证获取模型的置信度 86

5.5.4　模型的改进方案 88

5.6　小结 88

第 6 章　支持向量机 89

6.1　求解忠诚度映射问题 89

6.2　SVM 的推导过程 91

6.3　非线性数据 92

6.3.1　核技巧 92

6.3.2　软间隔 96

6.4　利用 SVM 进行情绪分析 97

6.4.1　类图 98

6.4.2　`Corpus` 类 99

6.4.3　从语料库返回一个无重复元素的单词集 102

6.4.4　`CorpusSet` 类 103

6.4.5　`SentimentClassifier` 类 107

6.4.6　随时间提升结果 111

6.5　小结 111

第 7 章　神经网络 113

7.1　神经网络的历史 113

7.2　何为人工神经网络 114

7.2.1　输入层 115

7.2.2　隐含层 116

7.2.3　神经元 117

7.2.4　输出层 122

7.2.5　训练算法 122

7.3 构建神经网络……125
7.3.1 隐含层数目的选择……126
7.3.2 每层中神经元数目的选择……126
7.3.3 误差容限和最大 epoch 的选择……126
7.4 利用神经网络对语言分类……127
7.4.1 为语言编写接缝测试……129
7.4.2 网络类的交叉验证……132
7.4.3 神经网络的参数调校……135
7.4.4 收敛性测试……136
7.4.5 神经网络的精度和查全率……136
7.4.6 案例总结……136
7.5 小结……136
第 8 章 聚类……137
8.1 用户组……138
8.2 *K* 均值聚类……139
8.2.1 *K* 均值算法……139
8.2.2 *K* 均值聚类的缺陷……140
8.3 EM 聚类算法……141
8.4 不可能性定理……142
8.5 音乐归类……142
8.5.1 数据收集……143
8.5.2 用 *K* 均值聚类分析数据……144
8.5.3 EM 聚类……146
8.5.4 爵士乐的 EM 聚类结果……149
8.6 小结……151
第 9 章 核岭回归……153
9.1 协同过滤……153
9.2 应用于协同过滤的线性回归……154
9.3 正则化技术与岭回归……157
9.4 核岭回归……158
9.5 理论总结……158
9.6 用协同过滤推荐啤酒风格……159
9.6.1 数据集……159
9.6.2 我们所需的工具……159
9.6.3 评论者……162
9.6.4 编写代码确定某人的偏好……164
9.6.5 利用用户偏好实现协同过滤……166
9.7 小结……167

第 10 章　模型改进与数据提取……169
10.1　维数灾难问题……169
10.2　特征选择……171
10.3　特征变换……173
10.4　主分量分析……175
10.5　独立分量分析……177
10.6　监测机器学习算法……179
10.6.1　精度与查全率：垃圾邮件过滤……179
10.6.2　混淆矩阵……181
10.7　均方误差……182
10.8　产品环境的复杂性……183
10.9　小结……183
第 11 章　结语……185
11.1　机器学习算法回顾……185
11.2　如何利用这些信息来求解问题……186
11.3　未来的学习路线……187
作者介绍……188
封面介绍……188

前言

这是一本介绍如何解决棘手问题的书。机器学习是计算技术的一项令人叹为观止的应用，因为它要解决的很多问题都源自科幻小说。机器学习算法可用于解决语音识别、映射、推荐以及疾病检测等复杂问题。机器学习的应用领域浩瀚无垠，也正因为如此它才这样引人入胜。

然而，这种灵活性也使得机器学习技术令人望而却步。它的确可以解决许多问题，但如何知晓我们求解的是否是正确的问题，或者是否应最先求解某个问题呢？此外，令人沮丧的是，大部分学术编码标准都不够严密。

即便是在今天，人们仍未对如何编写高质量的机器学习代码给予足够的关注，这是无比遗憾的。将一种观念在整个行业广泛传播的能力取决于有效沟通的能力。如果我们编写的代码本身就质量低劣，那么恐怕不会有很多人愿意聆听我们的讨论。

本书便是我对于该问题的回答。我试图按照容易理解的方式为大家讲授机器学习。这门学科本身难度就不小，况且我们还需要阅读代码，尤其是那些极难理解的古老 C 实现，无疑更是雪上加霜。

很多读者会对本书采用 Ruby 而非 Python 感到非常困惑。我的理由是用 Ruby 编写测试程序是一种诠释你所写代码的美妙方式。本质上，这本通篇采用测试驱动方法的书讲述的是如何沟通，具体说来是如何与机器学习这个奇妙的世界沟通。

从本书可学到的知识

应该说，本书对机器学习的介绍并不全面。因此，我向你强烈推荐 Peter Flach 编著的 *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*（Cambridge University Press）[1]。如果你希望读一些数学味道较浓的书，可参阅 Tom Mitchell 的 Machine

注 1：中文版即将由人民邮电出版社出版。——编者注

Learning 系列。此外，你还可参考 Stuart Russel 和 Peter Norvig 编写的有关人工智能的名著 *Artificial Intelligence: A Modern Approach*, 3rd Edition（Prentice Hall）。

阅读完本书之后，你并不会获得机器学习的博士学位，但我希望本书能向你传授足够的知识，帮助你开始研究如何利用机器学习技术解决涉及数据的现实问题。对于求解问题的方法以及如何在基础层面上使用它们，本书会提供大量相关示例。

你还将掌握如何解决那些比普通的单元测试更为模糊的复杂问题。

本书的阅读方法

阅读本书的最佳方法是找到一些能够让你感到兴奋的例子。我力图每一章都介绍一些这样的例子，虽然有时无法尽如人意。我不希望本书过于偏重理论，而是希望通过示例向你介绍机器学习能够解决的一些具体问题，以及我处理数据的方法。

在本书的大多数章中，我都试图在一开始便引入一个商业案例，之后开始研究一个可求解的问题，直到结尾。本书是一部短篇读物，因为我希望你能够专注于理解书中的代码，并深入思考这些问题，而非沉浸在理论当中。

本书读者

本书面向的读者包括三个群体：开发人员、CTO 以及商业分析师。

开发人员已经熟知如何编写代码，且想更多地了解机器学习这个激动人心的领域。他们拥有在计算环境中求解问题的背景，可能使用过 Ruby 编写代码，也可能从未接触过这种语言。他们是本书的主要读者群，但本书在写作时也兼顾了 CTO 和商业分析师。

CTO 是那些真正希望了解如何利用机器学习技术提升公司业务的人。他们可能听说过 K 均值算法、K 近邻算法，但不知道这些算法的适用性。商业分析师与之相似，只是不像 CTO 那样关注技术细节。为了这两个读者群，我在每章的开头都准备了一个商业案例。

作者联系方式

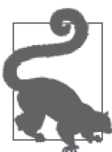
如果你喜欢我之前所做的演讲，或想为某个问题寻求帮助，欢迎通过电子邮件与我联系。我的邮箱是 matt@matthewkirk.com。为了增进我们之间的联系，如果你来到西雅图地区，且我们的日程允许的话，我非常乐意请你喝一杯咖啡。

如果希望查看本书的所有代码，可从 GitHub 站点 http://github.com/thoughtfulml 免费下载。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语或突出强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。



该图标表示非常重要的警告，请仔细阅读。

使用代码示例

补充材料（代码示例、练习等）可以从 http://github.com/thoughtfulml 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书

的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Thoughtful Machine Learning* by Matthew Kirk（O'Reilly). Copyright 2015 Matthew Kirk, 978-1-449-37406-8"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920032298.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

- Mike Loukides，对于我的利用测试驱动开发来编写机器学习代码的想法，他表现出了浓厚的兴趣。
- 我的编辑 Ann Spencer，在我撰写本书的数月间，她教给了我大量编辑知识和技巧，并针对本书的设计提出了宝贵意见。

我要感谢 O'Reilly 团队的所有成员，是他们成就了本书。我尤其要感谢下面这些人。

我的审稿人：

- Brad Ediger，当我提出要编写一本关于基于测试驱动的机器学习代码的书籍时，他对于这一古怪想法兴奋不已，并对本书的初稿提出了大量极有价值的意见。
- Starr Horne，他在审阅过程中向我提供了大量真知灼见。感谢他在电话会议中与我讨论了机器学习、错误报告等。
- Aaron Sumner，他针对本书的代码结构提出了极有价值的意见。

我极为出色的合作者，以及在本书编写过程中提供帮助的朋友们：Edward Carrel、Jon-Michael Deldin、Christopher Hobbs、Chris Kuttruff、Stefan Novak、Mike Perham、Max Spransy、Moxley Stratton 以及 Wafa Zouyed。

如果没有家人的支持和鼓励，我不可能顺利完成本书。

- 献给我的妻子 Sophia，是她帮助我坚持梦想，并帮助我将对本书的设想变成现实。
- 献给我的祖母 Gail，在我幼年时，是她引导我对学习产生了浓厚的兴趣。我还记得在一次公路旅行中，她心无旁骛地向我询问一些关于我正在看的那本“咖啡书”（实际上是讲 Java 的书）的问题。

- 献给我的父亲 Jay 和母亲 Carol，是他们教会了我如何分析系统，以及如何为其注入人类的情感。
- 献给我的弟弟 Jacob 以及侄女 Zoe 和 Darby，感谢他们教会了我如何通过孩子的眼光重新认识世界。

最后，我将本书献给科学和对知识不断求索的人。

第1章 测试驱动的机器学习

伟大的科学家既是梦想家，也是怀疑论者。在现代历史中，科学家们取得了一系列重大突破，如发现地球引力、登上月球、发现相对论等。所有这些科学家都有一个共同点，那就是他们都有着远大的梦想。然而，在完成那些壮举之前，他们的工作无不经过了周密的检验和验证。

如今，爱因斯坦和牛顿已离我们而去，但所幸我们处在一个大数据时代。随着信息时代的到来，人们迫切需要找到将数据转化为有价值的信息的方法。这种需求的重要性已日益凸显，而这正是数据科学和机器学习的使命。

机器学习是一门充满魅力的学科，因为它能够利用信息来解决像人脸识别或笔迹检测这样的复杂问题。很多时候，为完成这样的任务，机器学习算法会采用大量的测试。典型的测试包括提出统计假设、确定阈值、随着时间的推移将均方误差最小化等。理论上，机器学习算法具备坚实的理论基础，可从过去的错误中学习，并随时间的推移将误差最小化。

然而，我们人类却无法做到这一点。机器学习算法虽能将误差最小化，但有时我们可能“指挥失误”，没能令其将“真正的误差”最小化，我们甚至可能在自己的代码中犯一些不易察觉的错误。因此，我们也需要通过一些测试来发现自己所犯的错误，并以某种方式来记录我们的进展。用于编写这类测试的最为流行的方法当属*测试驱动开发*（Test-Driven Development，TDD）。这种“测试先行”的方法已成为编程人员的一种最佳实践。然而，这种最佳实践有时在开发环境中却并未得到运用。

采用驱动测试开发（为简便起见，下文中统称 TDD）有两个充分的理由。首先，虽然在主动开发模式中，TDD 需要花费至少 15%~35% 的时间，却能够排除多达 90% 的程序缺陷

（详情请参阅 http://research.microsoft.com/en-us/news/features/nagappan-100609.aspx）。其次，采用 TDD 有利于将代码准备实现的功能记录下来。当代码的复杂性增加时，人们对规格说明的需求也愈发强烈，尤其是那些需要依据分析结果制定重大决策的人。

哈佛大学的两位学者 Carmen Reinhart 和 Kenneth Rogoff 曾撰写过一篇经济学论文，大意是说那些所承担的债务数额超过其国内生产总值 90% 以上的国家的经济增长遭遇了严重滑坡。后来，Paul Ryan 在总统竞选中还多次引用了这个结论。2013 年，麻省大学的三位研究者发现该论文的计算有误，因为在其分析中有相当数量的国家未被考虑。

这样的例子还有很多，只是可能情况不像这个案例这样严重。这个案例说明，统计分析中的一处错误可能会对一位学者的学术声誉造成打击。一步出错可能会导致多处错误。上面两位哈佛学者本身都具有多年的研究经历，而且这篇论文的发表也经过了严格的同行评审，但仍然出现了这样令人遗憾的错误。这样的事情在任何人身上都有可能发生。使用 TDD 将有助于降低犯类似错误的风险，而且可以帮助这些研究者避免陷入万分尴尬的境地。

1.1 TDD的历史

1999 年，Kent Beck 通过其极限编程（extreme programming）方面的工作推广了 TDD。TDD 的强大源自其先定义目标再实现这些目标的能力。TDD 的实践步骤如下：首先编写一项无法通过的测试（由于此时尚无功能代码，因此测试会失败），再编写可使其通过的功能代码，最后重构初始代码。一些人依据众多测试库的颜色将其称为“红－绿－重构”（red-green-refactor）。红色表示编写一项最初无法通过的测试，而你需要记录自己的目标；绿色表示通过编写功能代码使测试通过。最后，对初始代码进行重构，直到自己对其设计感到满意。

在传统开发实践中，测试始终是中流砥柱，但 TDD 强调的是“测试先行”，而非在开发周期即将结束时才考虑测试。瀑布模型采用的是验收测试（acceptance test），涉及许多人员，通常是大量最终用户（而非开发人员），且该测试发生在代码实际编写完毕之后。如果不将功能覆盖范围作为考虑因素，这种方法看起来的确很好。很多时候，质量保证专业人员仅对他们感兴趣的方面进行测试，而非进行全面测试。

1.2 TDD与科学方法

TDD 之所以如此引人瞩目，部分原因在于它能够与人们及其工作方式保持良好的同步。它所遵循的“假设－测试－理论探讨”流程使之与科学方法有诸多相似之处。

科学需要反复试验。科学家在工作中也都是首先提出某个假设，接着检验该假设，最后将这些假设升华到理论高度。



我们也可将“假设 – 测试 – 理论探讨”这个流程称为“红 – 绿 – 重构”。

与科学方法一样，对于机器学习代码，测试（检验）先行同样适用。大多数机器学习践行者都会运用某种形式的科学方法，而 TDD 会强制你去编写更加清晰和稳健的代码。实际上，TDD 与科学方法的关系绝不止于相似。本质上，TDD 是科学方法的一个子集，理由有三：一是它需要构建有效的逻辑命题；二是它通过文档共享结果；三是它采用闭环反馈的工作机制。

TDD 之美在于你也可利用它进行试验。很多时候，首先编写测试代码时，我们都抱有一个信念，即最初测试中遇到的那些错误最终一定可被修正，但实际上我们并非一定要遵循这种方式。我们可以利用测试来对那些可能永远不会被实现的功能进行试验。对于许多不易解决的问题，按照这种方式进行测试十分有用。

1.2.1　TDD可构建有效的逻辑命题

科学家们在使用科学方法时，首先尝试着去求解一个问题，然后证明方法的有效性。问题求解需要创造性猜想，但如果没有严格的证明，它只能算是一种“信念”。

柏拉图认为，知识是一种被证明为正确的信念。我们不但需要正确的信念，而且也需要能够证明其正确的确凿证据。为证明我们的信念的正确性，我们需要构建一个稳定的逻辑命题。在逻辑学中，用于证明某个观点是否正确的条件有两种——必要条件和充分条件。

必要条件是指那些如果缺少了它们假设便无法成立的条件。例如，全票通过或飞行前的检查都属于必要条件。这里要强调的是，为确保我们所做的测试是正确的，所有的条件都必须满足。

与必要条件不同，充分条件意味着某个论点拥有充足的证据。例如，打雷是闪电的充分条件，因为二者总是相伴出现的，但打雷并不是闪电的必要条件。很多情形下，充分条件是以统计假设的形式出现的。它可能不够完善，但要证明我们所做测试的合理性已然足够充分了。

为论证所提出的解的有效性，科学家们需要使用必要条件和充分条件。科学方法和 TDD 都需要严格地使用这两种条件，以使所提出的一系列论点成为一个有机整体。然而，二者的不同之处在于，科学方法使用的是假设检验和公理，而 TDD 使用的则是集成和单元测试（参见表 1-1）。

表1-1：TDD与科学方法的比较

	科学方法	TDD
必要条件	公理	纯粹的功能测试
充分条件	统计假设检验	单元和集成测试

示例1：借助公理和功能测试完成证明

法国数学家费马于1637年提出著名的猜想[1]：对于大于2的任意整数 n，关于 a、b、c 的方程 $a^n + b^n + c^n$ 不存在正整数解。表面上看，这好像是一个比较简单的问题，而且据说费马声称他已经完成了证明。他在读书笔记中写道："我确信已发现了一种美妙的证法，可惜这里空白的地方太小，写不下。"

此后的358年间，该猜想一直未得到彻底证实。1995年，英国数学家安德鲁· 怀尔斯（Andrew Wiles）借助伽罗瓦（Galois）变换和椭圆曲线完成了费马大定理的最终证明。他长达100页的证明虽然称不上优雅，但每一步都经得起严格推敲。每一小节的论证都承前启后。

这100页证明中的每一步都建立在之前已被人们证明的公理和假设的基础之上，这与功能测试套件何其相似！用程序设计术语来说，怀尔斯在其证明中使用的所有公理和断言都可作为功能测试。这些功能测试只不过是以代码形式展现的公理和断言，每一步证明都是下一小节的输入。

大多数情况下，软件生产过程中并不缺少测试。很多时候，我们所编写的测试都是关于代码的随意的断言。许多情形下，为了使用以前的样例，我们只对打雷而不对闪电进行测试。即，我们的测试只关注了充分条件，而忽略了必要条件。

示例2：借助充分条件、单元测试和集成测试完成证明

与纯数学不同，充分条件只关心是否有足够的证据来支持某个因果关系。下面以通货膨胀为例来说明。自19世纪开始，人们已经在研究这种经济学中的神秘力量。要证明通货膨胀的存在，我们所面临的问题是并无任何公理可用。

不过，我们可以依据来自观察的充分条件来证明通货膨胀的确存在。我们观察过经济数据并从中分离出已知正确的因素，根据此经验，我们发现随着时间的推移，尽管有时也会下降，但长期来看经济是趋于增长的。通货膨胀的存在可以只通过我们之前所做的具有一致性的观察来证明。

在软件开发领域，经济学中的这类充分条件对应集成测试。集成测试旨在测试一段代码的首要行为。集成测试并不关心代码中微小的改动，而是观察整个程序，看所期望的行为是否能够如期发生。同样，如果将经济视为一个程序，则我们可断言通货膨胀或通货紧缩是存在的。

注1：即我们熟知的费马大定理。——译者注

1.2.2 TDD要求你将假设以文字或代码的形式记录下来

学术机构通常要求教授发表其研究成果。虽然有很多人抱怨各大学过于重视发表文章，但其实这样做是合理的：发表是一种使研究成果成为永恒的方法。如果教授们决定独自研究并取得重大突破，却不将其成果发表，那么这种研究将无任何价值。

TDD 同样如此：测试在同行评审中能够发挥重要的作用，也可作为一个版本的文档。实际上很多时候，在使用 TDD 时，文档并不是必需的。由于软件具有抽象性，且总处在变化之中，因此如果某人没有将其代码文档化或对代码进行测试，将来它便极有可能被修改。如果缺乏能够保证这些代码按特定方式运行的测试，则当新的编程人员参与到该软件的开发和维护工作中时，将无法保证他不会改动代码。

1.2.3 TDD和科学方法的闭环反馈机制

科学方法和 TDD 均采用闭环反馈的机制。当某人提出一项假设，并对其进行检验（测试）时，他会找到关于自己所探索问题的更多信息。对于 TDD，也同样如此；某个人对其所想进行测试，之后当他开始编写代码时，对于如何进行便可以做到心中有数。

总之，TDD 是一种科学方法。我们提出一些假设，对其进行检验（测试），之后再重新检视。TDD 践行者们遵循的也是相同的步骤，即首先编写无法通过的测试，接着找到解决方案，然后再对这个解决方案进行重构。

示例：同行评审

许多领域，无论是学术期刊、图书出版，还是程序设计领域，都有自己的同行评审，且形式各异。原因编辑（reason editor）之所以极有价值，是因为他们对于一部作品或一篇文章而言是第三方，能够给出客观的反馈意见。在科学界，与之对应的则是对期刊文章的同行评审。

TDD 则不同，因为第三方是一个程序。当某人编写测试时，程序以代码的形式表示假设和需求，而且是完全客观的。在其他人查看代码之前，这种反馈对于程序开发人员检验其假设是很有价值的。此外，它还有助于减少程序缺陷和功能缺失。

然而，这并不能缓解机器学习或数学模型与生俱来的问题，它只是定义了处理问题和寻求足够好的解的基本过程。

1.3 机器学习中的风险

对于开发过程而言，虽然使用科学方法和 TDD 可提供良好的开端，但我们仍然可能遇到一些棘手的问题。一些人虽然遵循了科学方法，但仍然得到了错误的结果；TDD 可帮助我们创建更高质量的代码，且更加客观。接下来的几个小节将介绍机器学习中经常会遇到的几个重要问题：

- 数据的不稳定性
- 欠拟合
- 过拟合
- 未来的不可预测性

1.3.1　数据的不稳定性

机器学习算法通过将离群点最少化来尽量减少数据中的不稳定因素。但如果错误的来源是人为失误，该如何应对？如果错误地表示了原本正确的数据，最终将使结果偏离真实情况，从而产生偏倚。

对我们所拥有的不正确信息的数量予以考虑，这是一个重要的现实问题。例如，如果我们使用的某个应用程序编程接口（API）将原本表示二元信息的 0 和 1 修改为 –1 和 +1，则这种变化对于模型的输出将是有害的。对于时间序列，其中也可能存在一些缺失数据。这种数据中的不稳定性要求我们找到一种测试数据问题的途径，以减少人为失误的影响。

1.3.2　欠拟合

如果模型未考虑足够的信息，从而无法对现实世界精确建模，将产生*欠拟合*（underfitting）现象。例如，如果仅观察指数曲线上的两点，我们可能会断言这里存在一个线性关系（如图 1-1 所示）。但也有可能并不存在任何模式，因为只有两个点可供参考。

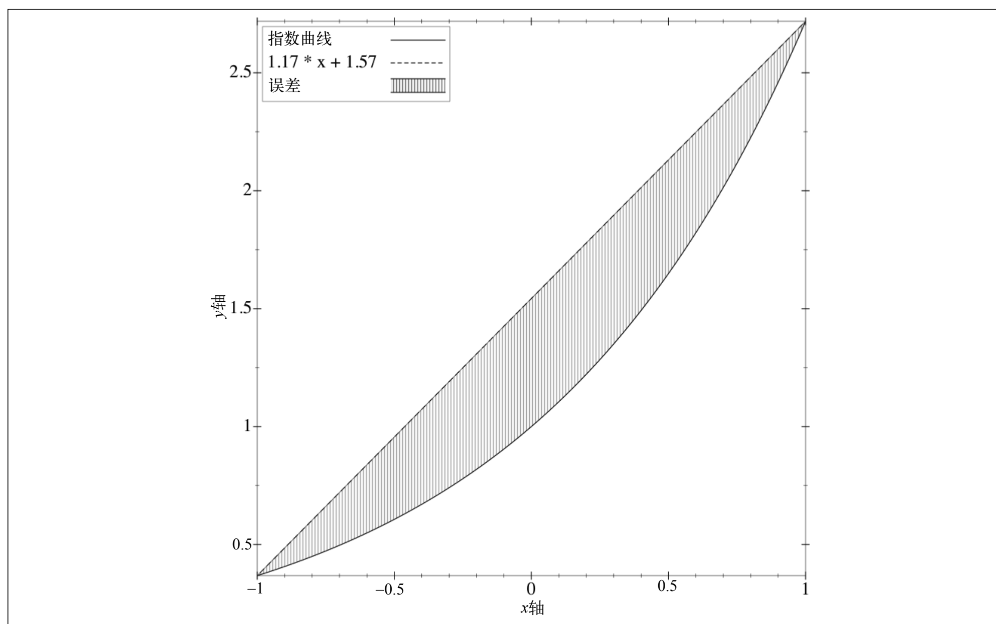


图 1-1：在 [−1,+1] 区间内，直线可对指数曲线取得良好的逼近效果

不幸的是，如果对该区间（[−1,+1]）进行扩展，将无法看到同样的逼近效果，取而代之的是显著增长的逼近误差（如图 1-2 所示）。

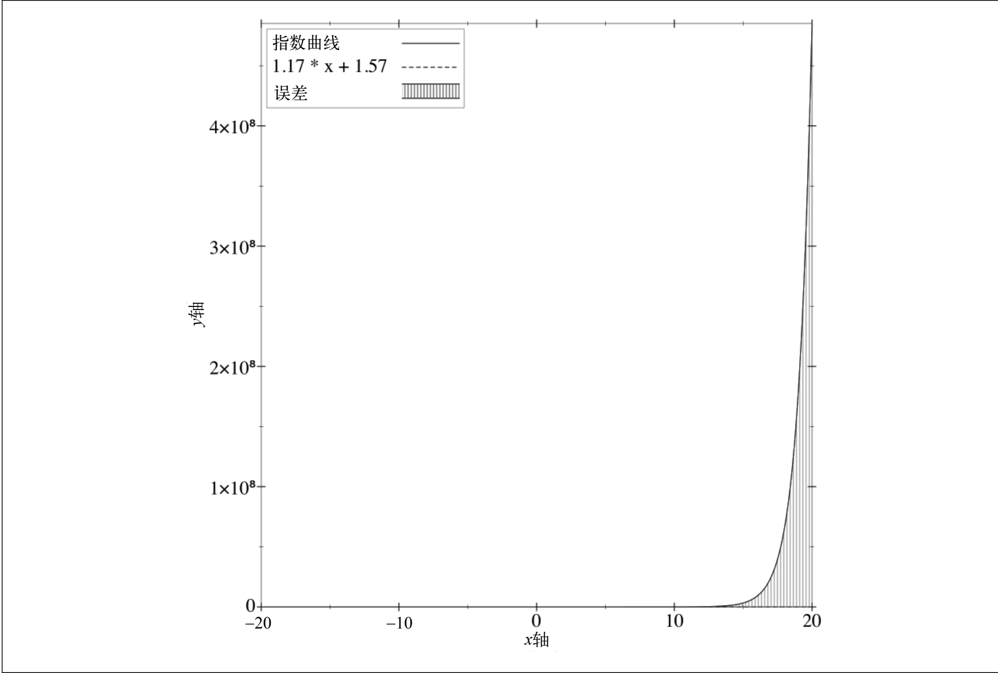


图 1-2：在 [−20,20] 区间内，直线将无法拟合指数曲线

统计学中有一个称为 power 的测度，它表示无法找到一个假负例（false negative）的概率。当 power 的值增大时，假负例的数量将减少。然而，真正影响该测度的是样本规模。如果样本规模过小，将无法获取足够的信息，从而无法得到一个良好的解。

1.3.3 过拟合

样本数太少是很不理想的一种情形，此时还存在对数据产生过拟合（overfitting）的风险。仍以相同的指数曲线为例，比如共有来自这条指数曲线的 30 000 个采样点。如果我们试图构建一个拥有 300 000 个算子的函数，便是对指数曲线过拟合，其实际上是记忆了全部 30 000 个数据点。这是有可能出现的，但如果有一个新数据点偏离了那些抽样，则这个过拟合模型对该点将产生较大的误差。

表面看来缓解模型欠拟合的最佳途径是为其提供更多的信息，但实际上这本身可能就是一个难以解决的问题。数据越多，通常意味着噪声越多，问题也越多。使用过多的数据和过于复杂的模型将使学习到的模型只能在该数据集上得到合理的结果，而对其他数据集将几乎完全不可用。

1.3.4 未来的不可预测性

机器学习非常适合不可预测的未来，因为大多数算法都需要从新息（即新的信息）中学习。但当新息到来时，其形式可能是不稳定的，而且会出现一些之前未预料到的新问题。我们并不清楚什么是未知的。在处理新息时，有时很难预测我们的模型是否仍能正常工作。

1.4 为降低风险应采用的测试

既然我们面临着若干问题，如不稳定的数据、欠拟合的模型、过拟合的模型以及未来数据的不确定性，到底应如何应对？好在有一些通用指导方针和技术（被称为启发式策略）可循，若将其写入测试程序，则可降低这些问题发生的风险。

1.4.1 利用接缝测试减少数据中的不稳定因素

在其著作 *Working Effectively with Legacy Code*（Prentice Hall 出版）中，Michale Feathers 在处理遗留代码时引入了接缝测试（testing seams）这个概念。接缝是指一个代码库的不同部分在集成时的连接点。在遗留代码中，很多时候都会遇到这样的代码：其内部机制不明，但当给定某些输入时，其行为可预测。机器学习算法虽不等同于遗留代码，但二者有相似之处。对待机器学习算法，也应像对待遗留代码那样，将其视为一个黑箱。

数据将流入机器学习算法，然后再从中流出。可通过对数据输入和输出进行单元测试来检验这两处“接缝”，以确保它们在给定误差容限内的有效性。

示例：对神经网络进行接缝测试

假设你准备对一个神经网络模型进行测试。你知道神经网络的输入数据取值需要介于 0 和 1 之间，且你希望所有数据的总和为 1。当数据之和为 1 时，意味着它相当于一个百分比。例如，如果你有两个小玩具和三个陀螺，则数据构成的数组将为 (2/5,3/5)。由于我们希望确保输入的信息为正，且和为 1，因此在测试套件中编写了下列测试代码：

```
it 'needs to be between 0 and 1' do
  @weights = NeuralNetwork.weights
  @weights.each do |point|
      (0..1).must_include(point)
  end
end

it 'has data that sums up to 1' do
  @weights = NeuralNetwork.weights
  @weights.reduce(&:+).must_equal 1
end
```

接缝测试是一种定义代码片段之间接口的好方法。虽然这个例子非常简单，但请注意，当

数据的复杂性增加时，这些接缝测试将变得更加重要。新加入的编程人员接触到这段代码时，可能不会意识到你所做的这些周密考虑。

1.4.2 通过交叉验证检验拟合效果

交叉验证是一种将数据划分为两部分（训练集和验证集）的方法，如图 1-3 所示。训练数据用于构建机器学习模型，而验证数据则用于验证模型能否取得期望的结果。这种策略提升了我们找到并确定模型中潜在错误的能力。

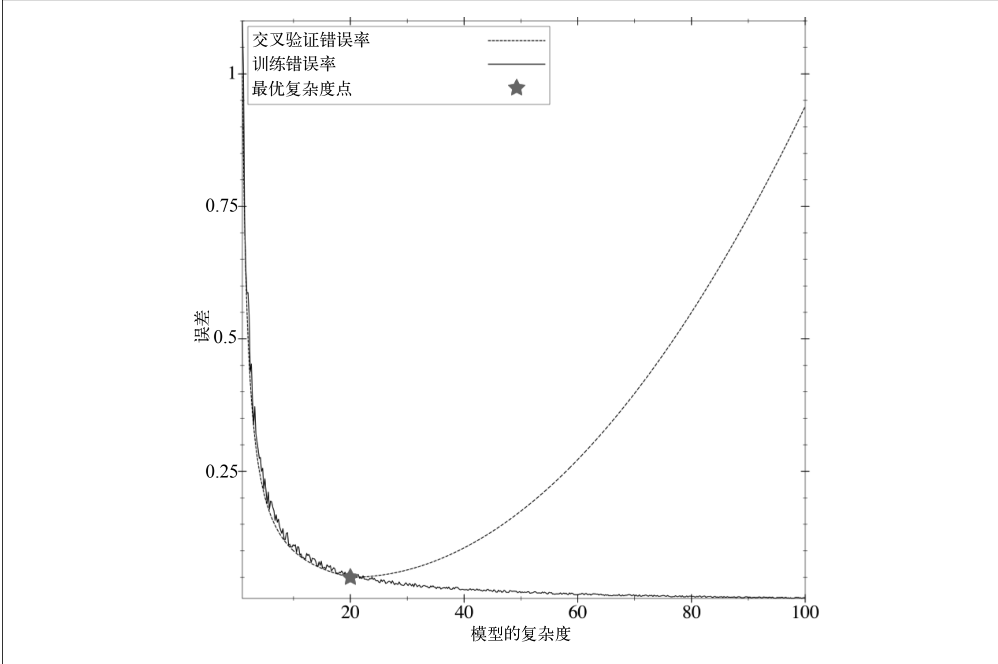


图 1-3：我们真正的目标是将交叉验证错误率或真实错误率最小化



训练专属于机器学习世界。由于机器学习算法的目标是将之前的观测映射为结果，因此训练非常重要。这些算法会依据人们所收集到的数据进行学习，因此如果缺少用于训练的初始数据集，该算法将百无一用。

交换训练集和验证集，有助于增加验证次数。你需要将数据集一分为二。第一次验证中，将集合 1 作为训练集，而将集合 2 作为验证集，然后将二者交换，再进行第二次验证。根据拥有的数据量，你可将数据划分为若干更小的集合，然后再按照前述方式进行交叉验证；如果你拥有的数据足够多，则可在任意数量的集合上进行交叉验证。

大多数情况下，人们会选择将验证数据和训练数据分为两部分，一部分用于训练模型，而

另一部分用于验证训练结果在真实数据上的表现。例如，假设你正在训练一个语言模型，利用隐马尔可夫模型（Hidden Markov Model，HMM）对语言的不同部分进行标注，则你会希望将该模型的误差最小化。

示例：对模型进行交叉验证

依据我们训练好的模型，训练错误率大致为 5%，但是当我们引入训练集之外的数据时，错误率可能会飙升到 15%。这恰恰说明了使用经划分的数据集的重要性；好比复式记账之于会计，对于机器学习而言这一点是极为必要的。例如：

```
def compare(network, text_file)
  misses = 0
  hits = 0

  sentences.each do |sentence|
    if model.run(sentence).classification == sentence.classification
      hits += 1
    else
      misses += 1
    end
  end

  assert misses < (0.05 * (misses + hits))
end

def test_first_half
  compare(first_data_set, second_data_set)
end

def test_second_half
  compare(second_data_set, first_data_set)
end
```

首先将数据划分为两个子集，这个方法消除了可能由机器学习模型中不恰当的参数引起的一些常见问题。这是在问题成为任何代码库的一部分之前，找到它们的绝佳途径。

1.4.3　通过测试训练速度降低过拟合风险

奥卡姆剃刀准则（Occam's Razor）强调对数据建模的简单性，并且认为越简单的解越好。这直接意味着"避免对数据产生过拟合"。越简单的解越好这种观点与过拟合模型通常只是记忆了它们的输入数据存在一些联系。如果能够找到更简单的解，它将发现数据中的一些模式，而非只是解析之前记忆的数据。

一种可间接度量机器学习模型复杂度的指标是它所需的训练时长。例如，假设你为解决某个问题，对两种不同的方法进行了测试，其中一种方法需要 3 个小时才能完成训练，而另一种方法只需 30 分钟。通常认为花费训练时间越少的那个模型可能越好。最佳方法可能是将基准测试包裹在代码周围，以考察它随着时间的推移变得更快还是更慢。

许多机器学习算法都需要设置最大迭代次数。对于神经网络，你可能会将最大 epoch 数设为 1000，表明你认为如果模型在训练中不经历 1000 次迭代，便无法获得良好的质量。epoch 这种测度所度量的是所有输入数据的完整遍历次数。

示例：基准测试

更进一步，你也可使用像 MiniTest 这样的单元测试框架。这类框架会向你的测试套件增加一定的计算复杂性和一个 IPS（iteration per second，每秒迭代次数）基准测试，以确保程序性能不会随时间而下降。例如：

```
it 'should not run too much slower than last time' do
  bm = Benchmark.measure do
    model.run('sentence')
  end
  bm.real.must_be < (time_to_run_last_time * 1.2)
end
```

这里，我们希望测试的运行时间不超过上次执行时间的 20%。

1.4.4 检测未来的精度和查全率漂移情况

精度（precision）和查全率（recall）是度量机器学习实现性能的两种方式。精度是对真正例的比例（即真正率）的度量 [1]。例如，若精度为 4/7，则意味着所预测的 7 个正例中共有 4 个样本是真正例。查全率是指真正例的数目与真正例和假负例数目之和的比率。例如，若有 4 个真正例和 5 个假负例，则相应的查全率为 4/9。

为计算精度和查全率，用户需要为模型提供输入。这使得学习流程成为一个闭环，并且由于数据被误分类后所提供的反馈信息，随着时间的推移，系统在数据上的表现也会得到提升。例如，网飞（Netflix）公司 [2] 会依据你的电影观看历史来预测你对某部影片的星级评价。如果你对该系统预测的评分不满意，并按照自己的意志重新评分，或者表明你对该部影片不感兴趣，则网飞再将你的反馈信息输入模型，以服务于将来的预测。

1.5 小结

机器学习是一门科学，并且需要借助客观的方法来解决问题。像科学方法一样，TDD 也有助于问题的解决。TDD 和科学方法之所以相似，是因为二者具有下列三个共同点。

- 二者均认为解应当符合逻辑，且具有有效性。

注 1：精度（precision）并不等同于真正率（true positive rate）。真正率是指实际正例中被预测正确的样本比例，而精度则是在所预测的正例中实际正例所占的比例。此外，精度也不同于准确率（accuracy），后者是指被正确分类的样本在整个测试集中所占的比例。——译者注

注 2：网飞公司是一家在线影片租凭提供商，拥有极为优秀的影片自动推荐引擎。——译者注

- 二者均通过文档共享结果，且可持续不断地工作。
- 二者都有闭环反馈的工作机制。

虽然科学方法和 TDD 有许多相似之处，但机器学习仍有其特有的问题：

- 数据的不稳定性
- 欠拟合
- 过拟合
- 未来的不可预测性

好在，借助表 1-2 所示的启发式策略，可在一定程度上缓解这些挑战。

表1-2：降低机器学习风险的启发式策略

问题/风险	启发式策略
数据的不稳定性	接缝测试
欠拟合	交叉验证
过拟合	基准测试（奥卡姆剃刀准则）
未来的不可预测性	随着时间的推移追踪精度和查全率

美妙的是，在真正开始编写代码之前，你可以编写或思考所有这些启发式策略。像科学方法一样，测试驱动开发也是求解机器学习问题的一种极有价值的方法。

第 2 章

机器学习概述

既然你选择了本书，说明你对机器学习感兴趣。对于何为机器学习你可能已有一定的了解，这是一门常常被模糊界定的学科。本章将介绍到底什么是机器学习，以及思考机器学习算法的一般框架。

2.1　什么是机器学习

机器学习是具有坚实理论基础的计算机科学和含噪声的真实数据的交集。本质上，机器学习是一门关于如何使机器像人类那样去理解数据的科学。

机器学习是一种人工智能，该领域中的算法或方法可从数据中提取出一些模式。一般说来，机器学习所要处理的问题不多，如表 2-1 所示。这些问题将在下面陆续进行介绍。

表2-1：机器学习问题

问题	机器学习类型
将数据拟合为某个函数或函数逼近	有监督学习
在无反馈条件下对数据进行推断	无监督学习
进行有奖励和回报的比赛或游戏	强化学习

2.1.1　有监督学习

有监督学习（或函数逼近）就是依据给定数据拟合出某种类型的函数。例如，给定图 2-1 所示的含噪声数据，可从中拟合出一条直线来逼近这些数据点。

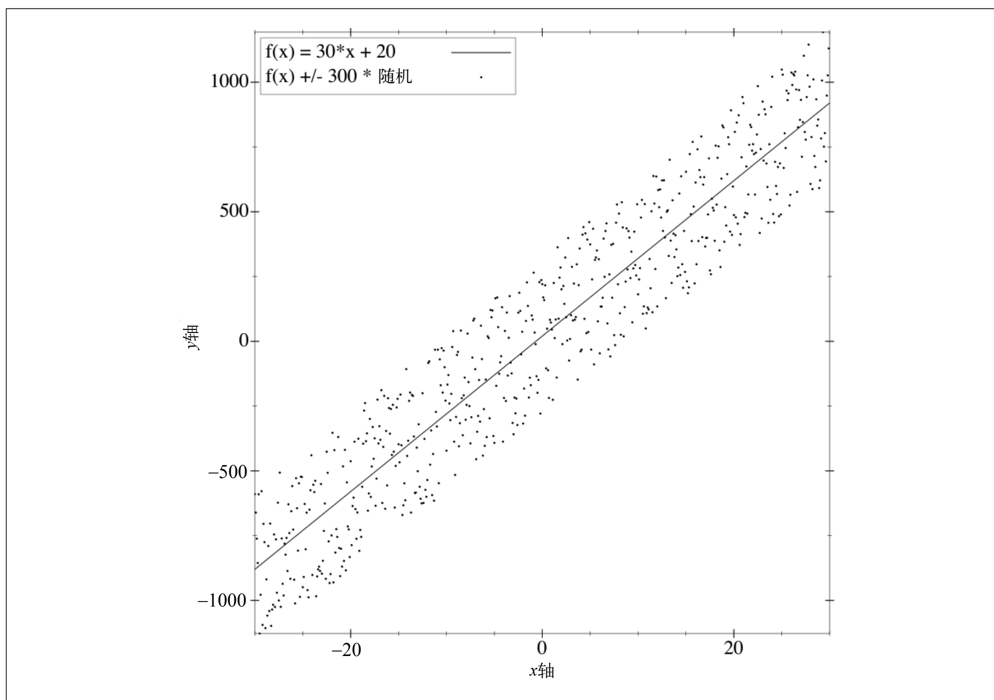


图 2-1：从一些随机数据点中拟合出的直线

2.1.2 无监督学习

无监督学习的目标是探索使得数据表现出特殊性的原因。例如，当给定许多数据点时，我们可依据相似性进行分组（如图 2-2 所示），或确定哪些变量更优。

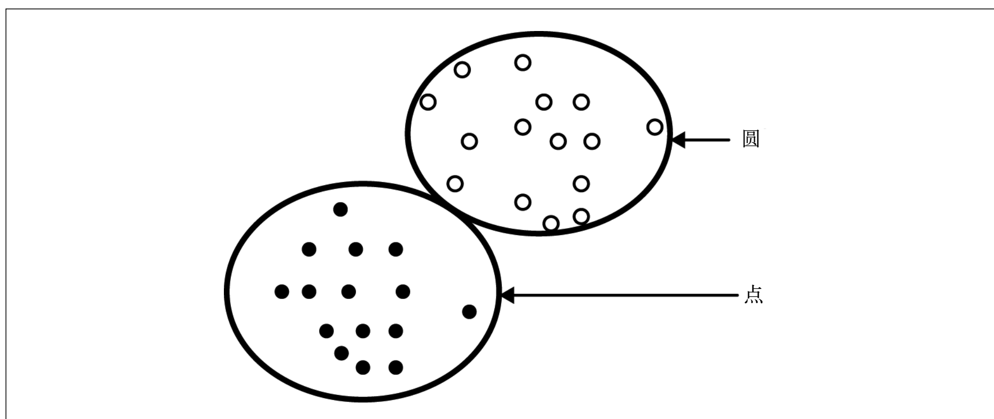


图 2-2：聚类是无监督学习的一个典型示例

2.1.3 强化学习

强化学习的目标是探索如何进行有奖励和回报的多阶段比赛或游戏。可将其视为一个对某物的生命周期进行优化的算法。强化学习算法的一个常见的例子是一只老鼠试图在迷宫中找到奶酪。在多数情况下，老鼠不会获得任何奖赏，除非它最终找到那块奶酪。

本书将只介绍有监督学习和无监督学习，而不涉及强化学习。如果你希望进一步了解强化学习，可参考最后一章中罗列的相关资源。

2.2 机器学习可完成的任务

真正使得机器学习独一无二的是其寻求最优解的能力。但每种机器学习算法都有其特殊性和缺点。针对特定的任务，总有一些算法的性能会优于其他算法。本书只介绍几种典型算法。表 2-2 展示了一个算法“矩阵”，以帮助你大致了解这些算法，并确定每种算法对你是否有用。

表2-2：机器学习算法矩阵

算法	类型	类	限制条件	适用场合
K 近邻（KNN）	有监督学习	基于实例的	一般说来，KNN 擅长度量基于距离的逼近，但易受维数灾难的影响	适合解决基于距离的问题
朴素贝叶斯分类（NB）	有监督学习	概率的	要求所给问题中，各输入相互独立	适用于问题域中各类的概率均大于 0 的情形
隐马尔可夫模型（HMM）	有监督 / 无监督学习	马尔可夫的	要求系统信息满足马尔可夫假设	适用于时间序列数据和无记忆性的信息
支持向量机（SVM）	有监督学习	决策面	要求待分类的两个类别有明确的差别	适用于求解两类分类问题
神经网络（NN）	有监督学习	非线性函数逼近	几乎没有限制条件	适用于二值输入
聚类	无监督学习	聚类	无限制	适用于当给定某种形式的距离度量（如欧氏距离、曼哈顿距离等）时，能表现出明确分组特性的数据
（核）岭回归	有监督学习	回归	限制很少	适用于变量连续的情形
滤波	无监督学习	特征变换	无限制	适用于有大量变量需要过滤的数据

在阅读本书时，请不时地回顾该表，以帮助你理解不同算法之间的联系。

只有将机器学习用于解决实际问题才能体现出其真正的价值，所以让我们开始实现一些机器学习算法吧！



在开始学习之前，你需要下载和安装 Ruby 编译器 https://www.ruby-lang.org/en/。我在撰写本书时使用的版本是 2.1.2，但考虑到 Ruby 社区极其活跃，版本更新十分迅速，因此我将所有的改动都将体现在本书配套的代码资源中，读者可从 GitHub 站点 https://github.com/thoughtfulml/examples 获取最新版本的代码。

2.3 本书采用的数学符号

本书借助数学知识来求解问题，但所有示例的设计都是面向程序开发人员的。本书中采用的数学符号如表 2-3 所示。

表2-3：本书示例中所采用的数学符号

符号	读法	功能
$\sum_{i=0}^{2} x_i$	从 x_0 到 x_2 的所有 x 之和	等价于 $x_0 + x_1 + \cdots + x_2$
$\lvert x \rvert$	x 的绝对值	无论 x 是否为正实数，$\lvert x\rvert$ 都为正实数。例如，当 $x=-1$ 时，$\lvert x\rvert=1$；当 $x=1$ 时，$\lvert x\rvert$ 仍为 1
$\sqrt{4}$	4 的平方根	是 2^2 的逆运算
$z_k = (0.5, 0.5)$	向量 z_k 的两个分量分别等于 0.5 和 0.5	表示 XY 平面上的一点，可用向量表示，向量是一组数值
$\log_2 2$	log2	它是方程 $2^i=2$ 的解
$P(A)$	事件 A 发生的概率	很多场合下，该值等于事件 A 发生的次数与所有事件发生次数总和的比值
$P(A \mid B)$	已知事件 B 发生，事件 A 发生的概率	等于 $\frac{P(A,B)}{P(B)}$
$\{1,2,3\} \cap \{1\}$	集合的交运算	结果为 {1}
$\{1,2,3\} \cup \{4,1\}$	集合的并运算	结果为 {1,2,3,4}
$\det C$	矩阵 $\boldsymbol{C}$ 的行列式	用于帮助确定某个矩阵是否可逆
$a \propto b$	a 与 b 成正比	意味着 $m \cdot a = b$
$\min f(x)$	最小化 $f(x)$	$f(x)$ 为将要最小化的目标函数
$\boldsymbol{X}^{\mathrm{T}}$	矩阵 $\boldsymbol{X}$ 的转置	交换矩阵行和列上的所有元素

2.4 小结

应当说，本章对机器学习的介绍并不全面，但并无大碍。对于像机器学习这样复杂的学科，我们总是需要不断地学习。本章已经为你学习后续章节奠定了良好的基础。

第 3 章

K 近邻分类

你很可能认识某个钟情于某个品牌（如某个技术公司或服装制造商）的人。通常，你可通过观察该人的衣着和言谈举止作出判断。但确定品牌亲和力是否还有其他方式呢?

对于一个电子商务网站，我们可通过查看相似用户的历史订单，了解他们曾购买过的商品，以判断品牌忠诚度。例如，我们不妨假设某个用户拥有一组历史订单，每个订单中包含两款商品，如图 3-1 所示。



图 3-1：涉及多个品牌的某用户历史订单

依据该用户的历史订单，可以看出他购买了大量 Milan Clothing Supplies（虚构的品牌，但不影响你理解该问题）的服装。在最近的五个订单中，他购买了五件 Milan Clothing Supplies 的衬衫。因此，可以说该用户对这家公司有某种程度的好感。对此有所了解后，如果我们提出一个问题：该用户对哪个品牌特别感兴趣？ Milan Clothing Supplies 无疑是首选。

这种一般性的思想被称为 *K* 近邻（KNN）分类算法。在本例中，*K*=5，而每个订单代表对某个品牌的投票。得到票数最多的品牌便是分类结果。本章将介绍 *K* 近邻分类，给出其定义，并通过一个示例来说明如何将其运用于检测人脸图像中是否有眼镜或胡须。



K 近邻分类是一种基于实例的有监督学习方法，尤其适用于对距离敏感的数据。它容易受到“维数灾难”的影响，而且与基于距离的算法一样，也面临许多其他问题，稍后我们将逐一进行介绍。

3.1 *K*近邻分类的历史

KNN 算法最早由 Drs. Evelyn Fix 和 J.L. Hodges Jr 博士在为美国航天医学空军学校（U.S. Air Force School of Aviation Medicine）撰写的一份未公开发表的技术报告中提出。Fix 和 Hodges 最初的研究重点是将分类问题划分为若干子问题。

- F 分布和 G 分布完全已知。
- 除少数参数外，F 分布和 G 分布完全已知。
- 除了概率密度函数可能存在外，F 和 G 均未知。

Fix 和 Hodges 指出，如果已知两类的分布，或除一些参数外分布已知，则可轻松得到一些有意义的解。因此，他们将工作的重点放在两类分布未知时如何分类这个更困难的问题上。他们的杰出工作为 KNN 算法奠定了坚实的基础。

该算法已被证明当数据规模趋于无穷时，其渐近错误率的上界为贝叶斯错误率的两倍。这意味着当更多的实体被添加到数据集中后，该算法的错误率将不会大于贝叶斯错误率。此外，KNN 的原理非常简单，很容易用于初步尝试某个分类问题的求解。对于许多情形，该算法都能够提供令人满意的结果。

但使用该算法也面临着一系列挑战。如，怎样选择 *K* 的值？如何确定哪些实例是近邻，而哪些不是？这些是我们将在接下来的几个小节中所要回答的问题。

3.2 基于邻居的居住幸福度

设想你准备购置一处房产，而且有了两个选择。你希望了解周围的邻居生活得是否幸福（你当然不希望搬到一个不幸福的居住区）。你去询问周围的人在那里生活得是否幸福，并将收集到的信息整理为表 3-1。

表3-1：房屋的幸福度

经度	纬度	是否幸福
56	2	是
3	20	否
18	1	是
20	14	否
30	30	是
35	35	是



之所以用经纬度来表示房屋的位置，是因为我们希望确定一个足够小的邻域。

假设我们感兴趣的两座房屋的位置分别是 (10,10) 和 (40,40)。那么选择哪一座房屋会有助于提升幸福感，选择哪一个会降低幸福感？确定这一点的一种方法是找到最近的邻居，然后看看他们是否幸福。这里的“最近”是指绝对距离（也称欧氏距离）意义上的最近。

如表 3-1 所示的二维点之间的欧氏距离的计算公式为 $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$。

整个分析流程可用下列 Ruby 代码表示：

```
require 'matrix'

# 向量v1和v2之间的欧氏距离
# 请注意Vector#magnitude表示从原点(0,0,....)到该向量首端的欧氏距离
distance = ->(v1, v2) {
  (v1 - v2).magnitude
}

house_happiness = {
  Vector[56, 2] => 'Happy',
  Vector[3, 20] => 'Not Happy',
  Vector[18, 1] => 'Happy',
  Vector[20, 14] => 'Not Happy',
  Vector[30, 30] => 'Happy',
  Vector[35, 35] => 'Happy'
}

house_1 = Vector[10, 10]
house_2 = Vector[40, 40]

find_nearest = ->(house) {
  house_happiness.sort_by {|point, v|
    distance.(point, house)
  }.first
```

```
}

find_nearest.(house_1) #=> [Vector[20, 14], "Not Happy"]
find_nearest.(house_2) #=> [Vector[35, 35], "Happy"]
```

以此为基础进行推断，可看到距离第一座房屋最近的邻居不幸福，而距离第二座房屋最近的邻居生活得很幸福。但如果增加考察的邻居数目，结果是否会有变化？

```
# 使用与上面相同的代码

find_nearest_with_k = ->(house, k) {
  house_happiness.sort_by {|point, v|
    distance.(point, house)
  }.first(k)
}

find_nearest_with_k.(house_1, 3)
#=> [[Vector[20, 14], "Not Happy"], [Vector[18, 1], "Happy"], [Vector[3, 20], "N
ot Happy"]]
find_nearest_with_k.(house_2, 3)
#=> [[Vector[35, 35], "Happy"], [Vector[30, 30], "Happy"], [Vector[20, 14], "No
t Happy"]]
```

增加参考的邻居数目并未改变分类结果！这是一件值得庆贺的好事，它提升了我们对分类结果的信心。该方法演示了 *K* 近邻分类过程。我们或多或少会参考最近的邻居，并依据他们的属性给出一个评分。在本例中，我们希望了解选择哪座房屋会更幸福，但数据的重要性绝对毋庸置疑。

由于其简单性（你刚才已经有所了解），KNN 是一种非常出色的算法，而且功能强大，可用于对数据进行分类或回归（详情请参阅下面的附注栏）。

分类与回归

请注意，在上面的场景中，我们只关心选择房屋是否会影响幸福感，即我们并不打算定量评估幸福感，而只是检查它是否符合我们的标准。这便是一个**分类问题**，而且可以多种形式出现。

很多时候，分类问题只涉及两个类别，这意味着只有两种可能的答案，如好或坏、真或假、正确或错误。许多问题都可转化为这种两类分类问题。

另一方面，我们也可为房屋寻找一个衡量幸福感的数值，这将是一个**回归问题**。本章不会介绍回归问题，不过在第 9 章讨论核岭回归时会介绍。

本章涉及众多知识点，并围绕 KNN 算法的使用分成数个主题。我们首先讨论如何选择 *K* 这个用于确定邻域的常量。之后再深入探讨何为“最近”，并给出一个利用 OpenCV 实现人脸分类的示例。

3.3 如何选择*K*

在评估房屋居住幸福度这个示例中，我们隐式地将 *K* 取为 5。对于依据某人最近的购物史作出快速判断来说，这个 *K* 值已经足够了。但对于更复杂的问题，我们可能无力猜测 *K* 具体取多少合适。

K 近邻算法中的 *K* 是介于 1 和数据点总数之间的一个任意整数。在这样宽泛的区间内，你可能会认为选择最优的 *K* 值非常困难，但实际上这个问题并不像你想象的那么复杂。要确定 *K*，主要有三种方案可供选择：

- 猜测
- 使用启发式策略
- 通过算法优化

3.3.1 猜测*K*的值

猜测是最简单的解决方案。在对商标分组的例子中，我们将 *K*=11 作为一个良好的猜测。我们知道，对于预测个体的购物行为，考察 11 份历史清单可能已经足够了。

很多时候，我们可定性地选择一个足够好的 *K* 来解决问题，因此猜测是可以奏效的。如果你希望用更科学的方法来确定 *K*，可采取某种启发式搜索策略。

3.3.2 选择*K*的启发式策略

有三种启发式策略可帮助你确定 KNN 算法中 *K* 的最优值。

- 当分类问题中只涉及两个类别时，不要将 *K* 取为偶数。
- *K* 的值应不小于类别总数加 1。
- 为避免出现噪声，*K* 的值应足够小。

1. 使*K*与类别总数互质

将 *K* 取为与类别总数互质的数，可保证投票数并列的情况较少出现。所谓互质是指两个数除 1 外再无其他公因子。例如，4 和 9 互质，而 3 和 9 非互质。

假设两类分类问题中所涉及的两个类别为“好”与“坏”。如果将 *K* 取为 6，则由于 6 是一个偶数，因此对于测试样本，最终可能得到票数并列的结果，如图 3-2 所示。

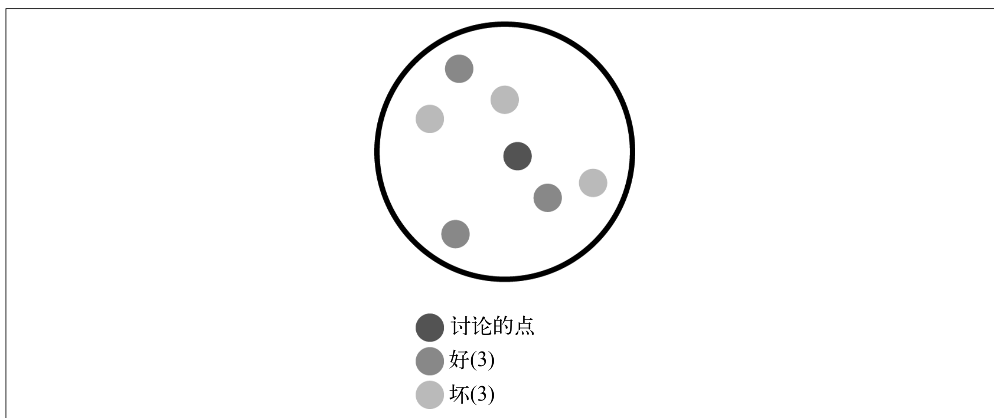


图 3-2：对于两类问题，当 K=6 时，可能出现得票数相等的情况

但是若将 K 取为 5（如图 3-3 所示），则将不会出现这种得票数相等的情况。

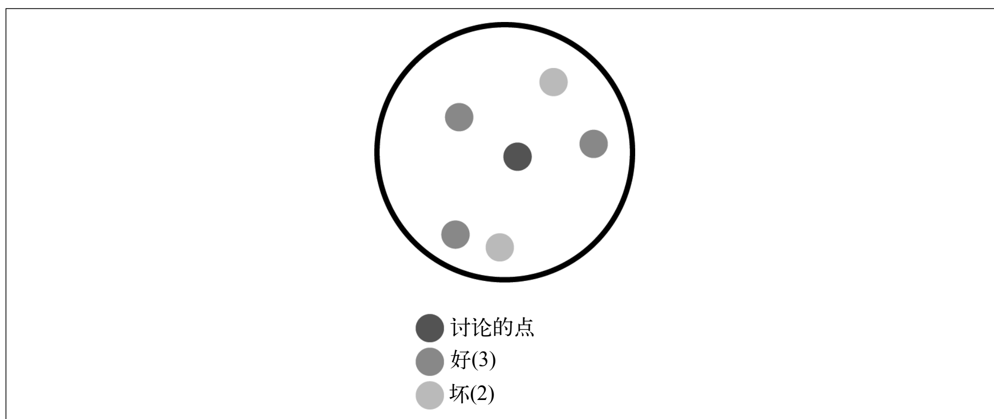


图 3-3：对于两类问题，若取 K=5，则不会出现得票数相等的情况

2. 将K取为不小于类别总数加1的值

假设现有三个类别：lawful、chaotic 以及 neutral。一种比较好的启发式策略是使 K 不小于 3，因为任何更小的数都绝无表示每一类别的可能。

为说明这一点，请参阅图 3-4，其中 K=2。

请注意上图中仅有两个类别得到了使用机会。这再次说明将 K 至少取为 3 的必要性。但依据我们从第一条启发式策略中的观察，票数并列并不是一个好现象。因此，我们实际上不应取 K=3，而应取 K=4（如图 3-5 所示）。

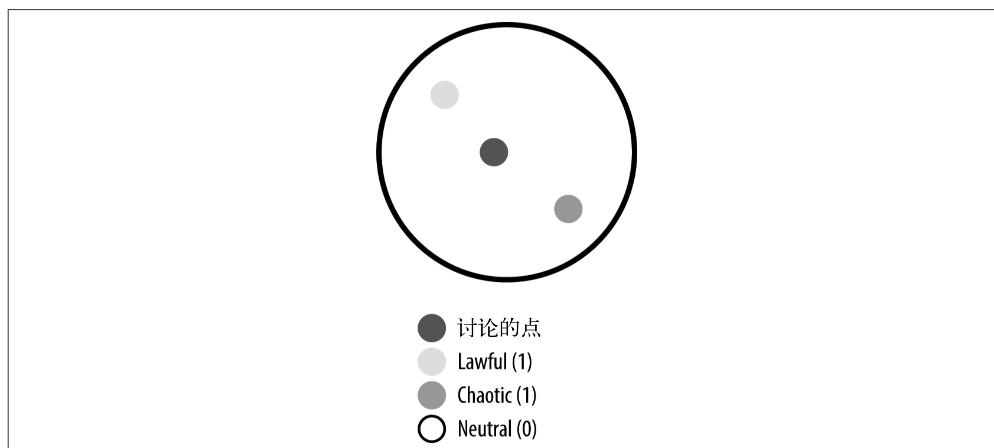


图 3-4：当 *K*=2 时，将无法表示全部三个类别

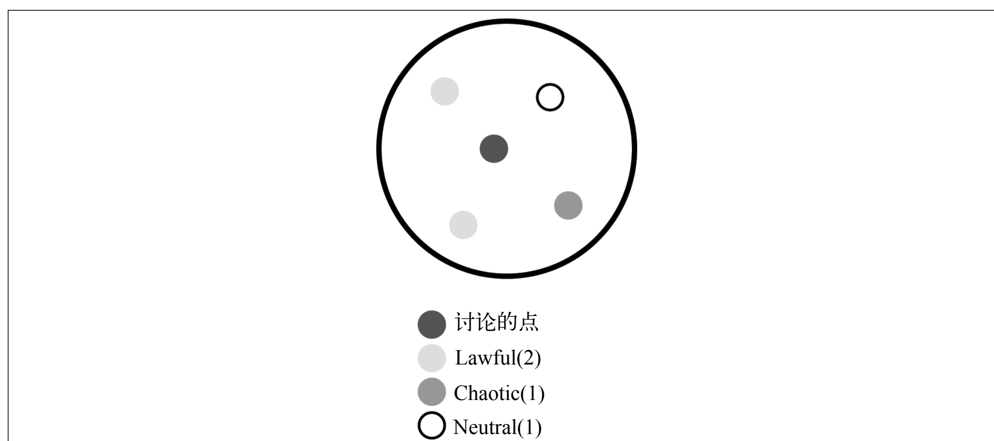


图 3-5：当 *K* 的值大于类别总数时，所有的类别均有被表示的机会

3. 选择足够小的*K*以避免出现噪声

可逐步增加 *K* 的值，直至达到整个数据集的规模。若把 *K* 取为整个数据集的容量，则分类结果对应出现频率最高的那个类。我们回到之前的品牌亲和力的例子，假设有 100 份订单，如表 3-2 所示。

表3-2：关于各品牌的订单数量

品牌	订单数量
Widget Inc.	30
Bozo Group	23
Robots and Rockets	12
Ion 5	35
总和	100

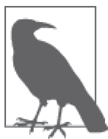
若取 K=100，则分类结果将为 Ion 5，因为它代表了订购历史的分布（即最常见的类）。这并非我们真正想要的，我们实际上希望确定最近的购买倾向。更具体地说，我们希望将进入分类环节的噪声数量降至最低。如果无法提出一种具体的算法，可将 K 设为一个较小的数值，如 K=3 或 K=11。

3.3.3　*K*的选择算法

前述选则 K 的方法多少有些定性，也不够科学，无怪乎有如此之多的算法专门用于在给定的训练集上优化 K 的取值。用于选择 K 的方法极多，包括了遗传算法和暴力网格搜索算法。

许多人认为 K 的选择应基于实现者所掌握的*领域知识*。例如，如果你了解当 K 取 5 时效果已经足够好了，则可直接选择这个值。

基于一个任意的 K 试图将误差最小化称为*爬山问题*（hill climbing problem）。其主要思想是对一组可能的 K 值轮流进行考察，直至找到一个可接受的误差。利用像遗传算法或暴力搜索这样的算法来寻求 K 的最优值的难点在于，当 K 增大时，分类的复杂性也相应增加，从而降低性能。换言之，当增加 K 时，程序的速度会逐渐变慢。



如果你希望详细了解如何将遗传算法用于对 K 寻优，请参考 Florian Nigsch 等在 *Journal of Chemical Information and Modeling* 期刊上发表的文章“Melting Point Prediction Employing k-Nearest Neighbor Algorithms and Genetic Parameter Optimization”（http://pubs.acs.org/doi/abs/10.1021/ci060149f）。

在我看来，从总体规模的 1% 开始迭代两次得到的 K 值已足够好了。通过对不同的 K 进行试验，对于什么样的值可用，什么样的不可用，你应有一个清晰的认识。

3.4　何谓“近”

假设你正位于某个城市街区的一角，那么你到该街区对角的距离有多远？

这个问题的答案取决于你施加的约束，即你是准备徒步穿越护栏，还是驾车前往？如果你采用的是后一种方案，则路程总长将为街区长度的两倍（曼哈顿距离，如图 3-6 所示）；反之，若你采用前一种方案直行，则路程总长为 $\sqrt{2x^2}$，其中 x 表示街区的长度（欧氏距离，如图 3-7 所示）。假定该街区的长度为 250 英尺（即 76.2 米），则我们可以说，乘车时，距离为 500 英尺（即 152.4 米），而步行时，距离约为 353.5 英尺（即 107.75 米）。

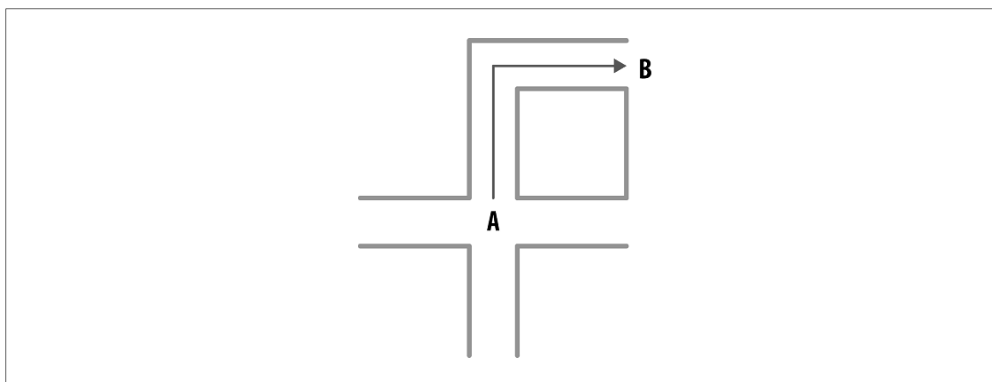


图 3-6：在某个街区中驱车从 A 点前往 B 点

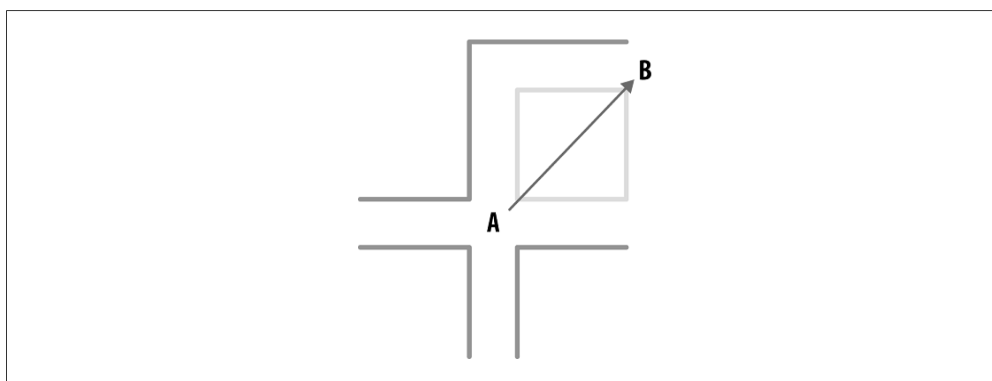


图 3-7：连接点 A 与点 B 的直线

你可能已经回忆起中学几何课上介绍过的有关知识。依据毕达哥拉斯定理（即勾股定理），若已知两直角边边长，则直角三角形斜边边长为 $\sqrt{a^2 + b^2}$。

按照现代数学的术语，这些距离称为度量（metric）。它们是一种点之间距离的测度。利用距离函数，可计算这些度量。在上面的例子中，距离函数分别为出租车距离（Taxicab distance）函数和欧氏距离函数。度量距离有多种方法，而选用何种距离度量对于理解 KNN 算法的工作机制至关重要，因为后者是基于数据之间的接近程度的。大部分情况下都会采用欧氏距离，它表示两点之间的最短路径。

3.4.1　Minkowski距离

对欧氏距离和出租车距离进行推广，可得到 Minkowski 距离。为了理解该距离度量，我们首先来看看出租车距离函数：

$$d_{\text{taxicab}}(x, y) = \sum_{i=1}^{n} \left| x_i - y_i \right|$$

该函数需要计算点 x 和 y 在所有维度上的绝对差。下面再来看看欧氏距离函数：

$$d_{\text{euclid}}(x,y)=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

请注意，由于上式中平方运算的结果为非负，且 $\sqrt{x}=x^{\frac{1}{2}}$，故可将上式整理为：

$$d_{\text{euclid}}(x,y)=\left(\sum_{i=1}^{n}|x_i-y_i|^2\right)^{\frac{1}{2}}$$

可以看出，此时它与上述出租车距离函数非常相似。Minkowski 将上述两个距离公式推广为如下形式：

$$d(x,y)=\left(\sum_{i=1}^{n}|x_i-y_i|^p\right)^{\frac{1}{p}}$$

通过引入一个新的参数 p，使得上述两种距离均成为 Minkowski 距离的特例，即当 p=1 时，Minkowski 距离特化为出租车距离，而当 p=2 时，Minkowski 距离则特化为欧氏距离。这非常耐人寻味，因为我们可根据需要任意增大 p 的值。虽然我们不打算探讨所有版本的 Minkowski 距离的应用，但相信你已具备了进一步学习的基础。

3.4.2 Mahalanobis距离

Minkowski 类型的距离函数存在的一个问题是，它们假定数据分布本质上应当具有对称性，即距离在所有方向上都是相同的。

很多时候，数据并不符合球状分布，因此不宜采用像 Minkowski 距离这样的对称距离。例如，在图 3-8 所示的情形中，我们应当对数据分布呈椭圆形这个特点予以考虑。像图中所示那样围绕数据画出一个标准圆是不可取的，我们需要选择一个能够更好地体现数据分布特点的形状。

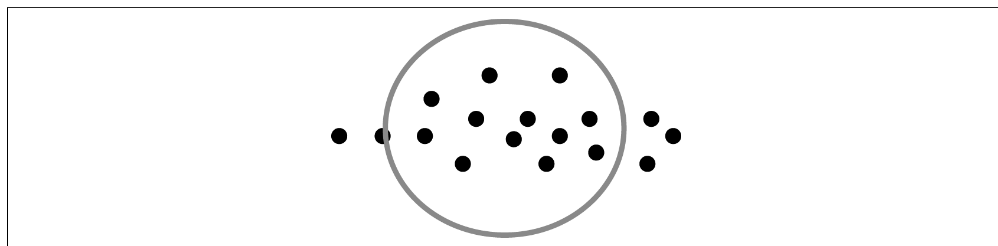


图 3-8：对于呈椭圆形分布的数据，利用 Minkowski 距离无法得到令人满意的结果

Mahalanobis 距离函数会考虑数据在每个维度上的波动性（参见图 3-9）。因此，对于数据

的每个维度，都存在一个 s_i，表示该数据集在此维度上的标准差的变量。

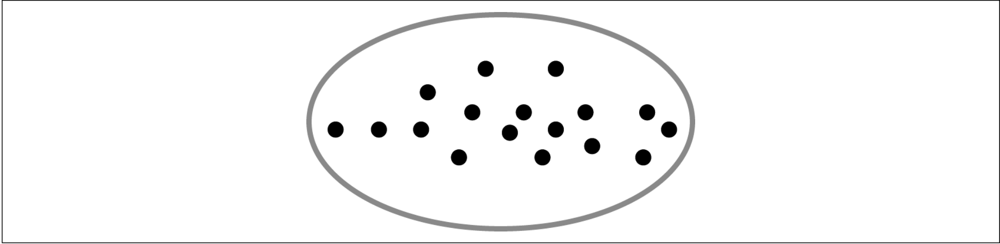


图 3-9：使用 Mahalanobis 距离

Mahalanobis 距离的计算公式如下所示：

$$d(x,y)=\sqrt{\sum_{i=1}^{n}\frac{(x_i-y_i)^2}{s_i^2}}$$

不难看出，该公式与欧氏距离非常类似，区别仅在于是否考虑数据集在各维度上的标准差。

3.5 各类别的确定

问题域中的类别可以相当宽泛。有时，所设置的各类别并不像我们最初想的那样完全互斥。因此，在构建 KNN 分类工具时需要特别注意的是，当模型中涉及的属性数目增加时，类的总数也呈指数级增长。

例如，若有两种属性颜色：红色和黄色，我们可得到三个类别，即红色、黄色以及橙色（参见图 3-10）。

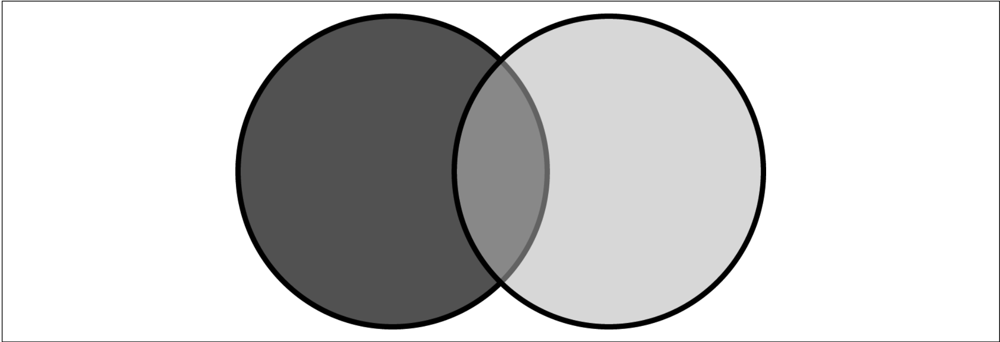


图 3-10：两种属性的混合

此时，如果我们再增加一个属性蓝色，将得到红色、黄色、蓝色、绿色、深褐色、橙色以及紫色共 7 个类别（参见图 3-11）。在第一种情况下，两个属性产生了 3 个类别，而在第

二种情形下，三个属性共产生了 7 个类别。

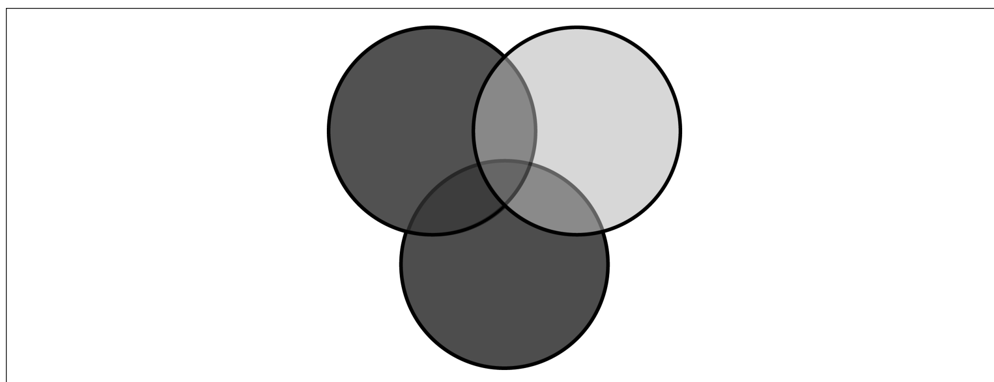


图 3-11：三种属性的混合

对于涉及 n 个属性的一般情形，它们可组合成的互斥类别数将为 2^n-1。

除非有充分的理由将混合类剔除，否则区分这一点是非常重要的。若共有 4 种属性，则可假定共需考虑 15 个类别，因此可将 *K* 取为一个不小于 16 的数。

维数灾难

K 近邻算法的缺点之一易受**维数灾难**（curse of dimensionality）的影响。所谓维数灾难是指维数越高，数据越稀疏，不同数据之间的距离也越远。可以想象一下一颗子弹飞出枪膛，其微粒随着时间逐渐在空气中扩散的过程。这个问题在基于局部性以及确定某些对象接近程度的算法中十分常见。

图 3-12 演示了当将单位球（即球心位于原点，半径为 1 的球）收缩为一个二维平面后，点之间的平均距离低于之前的平均水平。而当升维之后，会出现相反的情形。

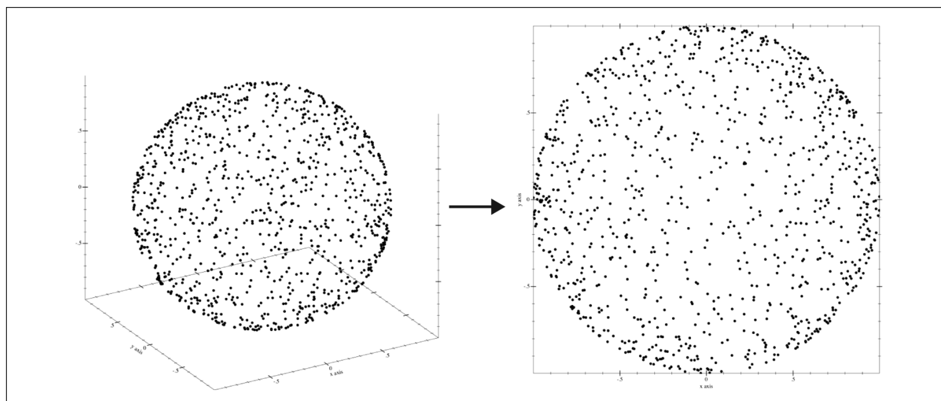


图 3-12：球面上的维数灾难

> *K* 近邻算法自身无法克服这个缺陷，必须借助其他手段才能降低这种风险，详情请参阅第 6 章。
>
> 在第 9 章和第 10 章中，我们还将进一步探讨维数灾难。

3.6　利用KNN算法和OpenCV实现胡须和眼镜的检测

假定你希望以一般的准确率来检测某人是否有胡须以及他是否佩戴了眼镜。如何完成这样的任务？对于这种数据的先验分布，我们实在知之甚少，因此结合计算机视觉开源库 OpenCV（Open Computer Vision）来使用 KNN 是一个不错的选择。下面首先介绍该程序的类框图，然后开始深入探讨如何从一幅含有人的图像中检测出人脸区域，之后将从这些人脸图像中提取一些特征。当特征数量足够多时，便可利用 KNN 构建一个人脸的邻域，以帮助我们确定输入图像的属性。



安装说明

本例的所有代码都可从 GitHub 站点 https://github.com/thoughtfulml/examples/tree/master/2-k-nearest-neighbors 获取。

由于 Ruby 是一种不断更新的语言，源码包中附带的 README 文件中包含了如何使用这些代码的最新说明。

你需要事先安装 ImageMagick、OpenCV 以及最新版的 Ruby 编译器。

3.6.1　类图

本例的基本流程是首先读入一幅原始图像（`Image` 类的实例），从中提取一幅较小的人脸图像（`Face` 类的实例），然后将所有的 `Face` 类实例存入一个 `Neighborhood` 类的实例中，后者中包含了很多带有标注信息的人脸图像（即 `Face` 类的实例）。完整的类图请参见图 3-13。

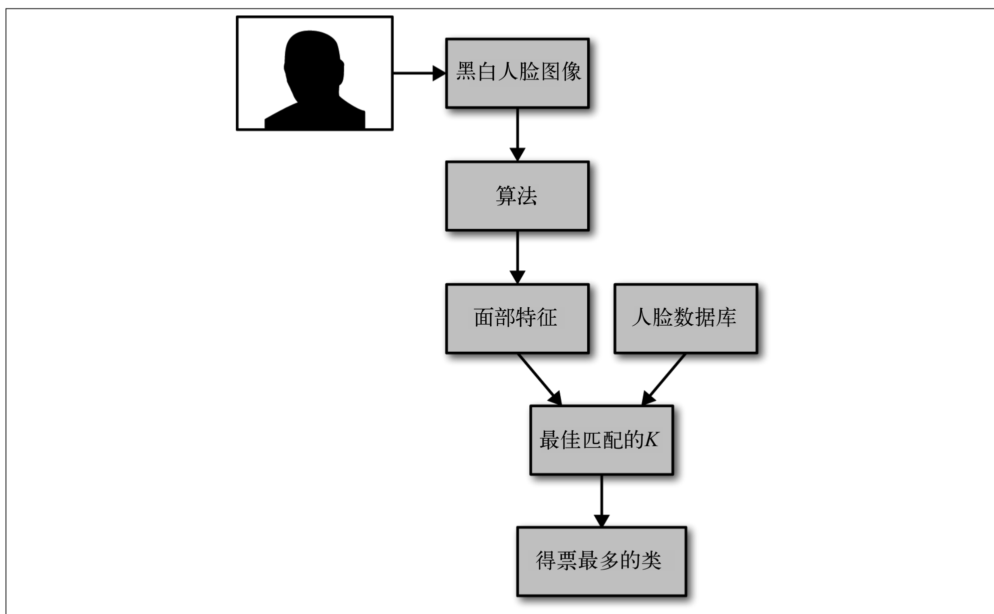


图 3-13：胡须和眼镜的检测器类图

3.6.2 从原始图像到人脸图像

`Image` 类先接收一幅含有人的图像，然后试图从中检测到人脸区域。我们可借助 OpenCV 来实现该功能。我们希望的输入和输出如图 3-14 和图 3-15 所示。

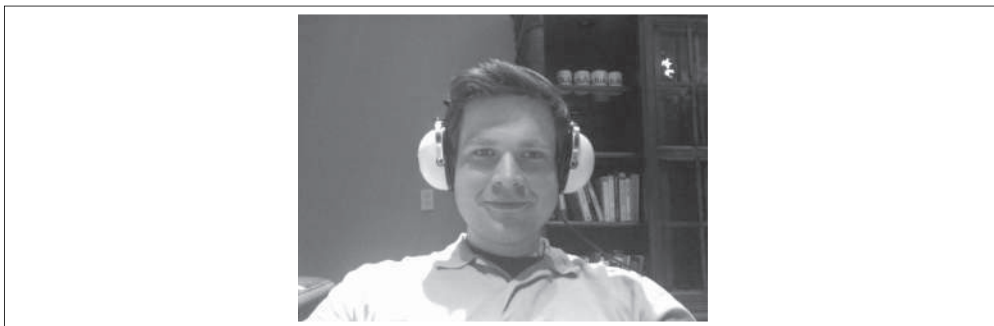


图 3-14：原始图像



图 3-15：利用 Haar 分类器提取到的人脸区域

如果你对 OpenCV 略有了解，便会想到利用类 Haar 特征来提取与人脸相似的区域。可利用 OpenCV 库提供的数据（即人脸检测训练结果）及检测函数来实现我们所需的功能。



这些数据可免费获取。实际上，这些数据并非我生成的。它们是 OpenCV 的某些贡献者在某个用于训练人脸检测器的训练集上训练得到的模型参数，而用于从人脸区域提取特征的功能模块也是由其他人贡献的。

OpenCV 与类 Haar 特征

利用 OpenCV 中的类 Haar 特征提取模块，可从一幅尺寸较大的含人脸的图像中检测出人脸区域，即得到一个包围人脸区域的矩形。依据上述信息可知，与背景元素不同，不同的人脸通常共有某些特征，因此我们可利用类 Haar 特征的训练信息来确定人脸区域对应的矩形。

如果你希望更深入地了解 OpenCV，可参考下列资源。

- Daniel Lélis Baggio 编著的 *Mastering OpenCV Practical Computer Projects*（Packt Publishing）。
- OpenCV 的自带文档（http://docs.opencv.org）也是极佳的学习材料。

为确定所得到的人脸区域是否正确，可使用 pHash。不同于 MD5 或 SHA1（这些都是密码散列），pHash 是一种利用海明距离（Hamming distance）来寻找最接近的匹配的感知（perceptual）散列。因此，即使该照片中的人脸位置有一些偏移量，其散列值仍与之前的吻合。



海明距离是指两等长字符串中对应位置上不同字符的个数。例如，对于字符串“apple”和“oople”，由于它们只有前两个字符不同，因此二者的海明距离为 2。注意，海明距离只适用于长度相等的字符串。

首先，需要对人脸检测方法进行测试，看它是否每次都返回相同的结果。我们对图 3-14 进行检测，并观察检测结果是否与图 3-15 一致：

```
# test/lib/image_spec.rb
require 'spec_helper'

describe Image do
  it 'tries to convert to a face avatar using haar classifier' do
    @image = Image.new('./test/fixtures/raw.jpg')
    @face = @image.to_face

    avatar1 = Phashion::Image.new("./test/fixtures/avatar.jpg")
    avatar2 = Phashion::Image.new(@face.filepath)
```

```
    assert avatar1.duplicate?(avatar2)
  end
end
```

可以预见上述代码一定会失败，因为 `@image.to_face` 没有任何实质内容，而且 `@face` 也缺少一个与之关联的文件路径。

为填补这些空白，可增加下列代码：

```
# lib/image.rb
# 检测人脸
Face = Struct.new(:filepath)

class Image
  HAAR_FILEPATH = './data/haarcascade_frontalface_alt.xml'
  FACE_DETECTOR = OpenCV::CvHaarClassifierCascade::load(HAAR_FILEPATH)

  attr_reader :filepath

  def initialize(filepath)
    @filepath = filepath
  end

  def self.write(filepath)
    yield
    filepath
  end

  def face_region
    @image = OpenCV::CvMat.load(@filepath, OpenCV::CV_LOAD_IMAGE_GRAYSCALE)
    FACE_DETECTOR.detect_objects(@image).first
  end

  def to_face
    name = File.basename(@filepath)
    outfile = File.expand_path("../../public/faces/avatar_#{name}", __FILE__)

    self.class.write(outfile) do
      image = MiniMagick::Image.open(@filepath)
      image.crop(crop_params)
      image.write(outfile)
    end

    Face.new(outfile)
  end

  def x_size
    face_region.bottom_right.x - face_region.top_left.x
  end

  def y_size
    face_region.bottom_right.y - face_region.top_left.y
```

```
  end

  def crop_params
    crop_params = <<-EOL
      #{x_size - 1}x#{y_size-1}+#{face_region.top_left.x + 1}+#{face_region.top_
left.y + 1}
    EOL
  end
end
```

可以看到，在上述代码中我们使用了两个库——MiniMagick 和 OpenCV。此外，还用到了来自 OpenCV 库的用于检测人脸的训练集 haarcascade_frontalface_alt.xml。此时，测试可以通过，并可认为 `Image` 类能够如期工作。但现在我们需要关注如何构建 `Face` 类。

特征、维度及实例

特征、**维度**以及**实例**是机器学习中使用频率极高的三个术语。

特征代表了给定数据集的一种属性。通常，特征由那些最重要的维度组合而成，其中每个维度都代表了对数据不同方面的刻画；而实例则表示具体的数据片段。

特征与维度之间的关系可用房间中的照明来类比（参见表 3-3）。假设共有三盏灯，则共有 8 种照明方案。但你真正希望了解的是采用哪种方案能够使房间足够明亮（这意味着至少应点亮两盏灯）。

表3-3：房间中的照明

第1盏灯是否被点亮	第2盏灯是否被点亮	第3盏灯是否被点亮	房间是否足够明亮
否	否	否	否
是	否	否	否
否	是	否	否
否	否	是	否
是	是	否	是
否	是	是	是
是	否	是	是
是	是	是	是

本例中只涉及了一种特征，即房间是否足够明亮，该特征基于描述三盏灯开启状态的三个维度。而实例则是这些灯开启状态的组合。

3.6.3 Face类

`Face` 类只负责完成一项功能，即加载一幅含有人脸的图像，并从中提取特征。随后，这些特征将送入 `Neighborhood` 类（将在下一节进行介绍）。从这里开始，情况开始变得复杂，因为特征可按照三种不同的方式来提取。

- 提取每个像素的明暗值（由于我们选用的图像为灰度图，因此每个像素所包含的是灰度值而非彩色值）。
- 使用 SIFT 算法。
- 使用 SURF 算法。

将图像的明暗表示为像素矩阵是一种简单直观的方法，但这种表示法会使我们陷入维数灾难。这种简单表示方法的代价是无法将那些无意义的噪声像素数目减少。有些算法能够直接处理灰度输入，如神经网络（将在第 7 章进行介绍）。例如，深度学习（deep learning）便利用了神经网络的这个优点，能够从灰度像素中直接提取出有意义的特征。

另一种方法是使用 SIFT（Scale Invariant Feature Transform，尺度不变的特征变换）算法。该算法是一种用于检测显著特征的计算机视觉算法，由 David Lowe 教授于 1999 年提出。该算法已被英属哥伦比亚大学申请了专利。相比于像素矩阵，该方法是一个巨大的进步。

还有一种与 SIFT 类似的算法，即 SURF（Speeded Up Robust Features，加速的稳健特征）。SURF 算法由 Herbert Bay 于 2006 年提出，是对 SIFT 的一种改进，目的在于提高计算效率。该算法已被证明可成功应用于识别图像的多种特征。

无论 SIFT 还是 SURF 都是很好的选择。OpenCV 提供了一个高质量的 SURF 版本，因此我们可利用它来从人脸图像中提取特征。

测试Face类

SURF 可为 `Face` 类提供两种信息片段，即关键点和描述符。关键点是指特征点在图像坐标系中的二维坐标 (x,y)，而描述符更有趣，它所包含的是从特征点的某个固定大小的邻域中计算出的 64 维或 128 维局部特征。我们不打算对 OpenCV 中的 SURF 实现的特征检测质量进行测试，而是需要确保数据始终一致，即所构造的两个 `Face` 类实例从同一幅图提取出的特征相同。

为此，可使用下列测试代码：

```
# test/lib/face_spec.rb
require 'spec_helper'
require 'matrix'

describe Face do
  let(:avatar_path) { './test/fixtures/avatar.jpg' }

  it 'has the same descriptors for the exact same face' do
    @face_descriptors = Face.new(avatar_path).descriptors
    @face2_descriptors = Face.new(avatar_path).descriptors

    @face_descriptors.sort_by! { |row| Vector[*row].magnitude }
    @face2_descriptors.sort_by! { |row| Vector[*row].magnitude }
```

```
    @face_descriptors.zip(@face2_descriptors).each do |f1, f2|
      assert (0.99..1.01).include?(cosine_similarity(f1, f2)),
        "Face descriptors don't match"
    end
  end

  it 'has the same keypoints for the exact same face' do
    @face = Face.new(avatar_path)
    @face2 = Face.new(avatar_path)

    # 这纯粹是因为Ruby的OpenCV实现中对SurfPoints缺少==表示
    @face.keypoints.each_with_index do |kp, i|
      f1 = Vector[kp.pt.x, kp.pt.y]
      f2 = Vector[@face2.keypoints[i].pt.x, @face2.keypoints[i].pt.y]

      assert (0.99..1.01).include?(cosine_similarity(f1, f2)),
        "Face keypoints do not match"
    end
  end
end
```

余弦相似度

你可能已经注意到，在上述代码中我们使用了一个名为 cosine_similarity 的函数。这也是确定两个对象之间接近程度的一种良好方式。我们采取的准则并不是看数据是否 100% 相等，而是看两个向量是否足够接近。

可在文件 spec_helper.rb 内写出该方法的实现，如下所示：

```
# test/spec_helper.rb
def cosine_similarity(array_1, array_2)
  v1 = Vector[*array_1]
  v2 = Vector[*array_2]
  v1.inner_product(v2) / (v1.magnitude * v2.magnitude)
end
```

对于比较两不同向量之间是否相似，余弦相似度是一种非常有用的测度。它实际上度量的是这两个向量之间的夹角，而非长度上的差异。例如，在本例中，若有向量 [1,1] 和 [2,2]，则可以看出，二者的余弦相似度为 1：

```
cosine_similarity([2,2], [1,1]) #=> 0.9999 ~ 1
```

由于描述符也是以向量的形式存在，因此在比较两描述符（比如人脸）相似性的场合，也可使用余弦相似度，至少可以考察它们的方向是否相同。

上面的 Face 类是用 Struct 类创建的一个动态类，因此我们需要为其填入一些代码片段：

```
# lib/face.rb

class Face
  include OpenCV
```

```
  MIN_HESSIAN = 600

  attr_reader :filepath

  def initialize(filepath)
    @filepath = filepath
  end

  def descriptors
    @descriptors ||= features.last
  end

  def keypoints
    @keypoints ||= features.first
  end

  private
  def features
    image = CvMat.load(@filepath, CV_LOAD_IMAGE_GRAYSCALE)
    param = CvSURFParams.new(MIN_HESSIAN)
    @keypoints, @descriptors = image.extract_surf(param)
  end
end
```

可以看到，该类除了提取特征点和描述符外，并无太多实质内容。MIN_HESSIAN 是一个推荐在 400~800 范围内选取的参数。当 MIN_HESSIAN 的值增加时，SURF 检测到的特征数目将减少，但这些特征的重要性将更为突出。同时，我们还从每幅人脸图像中提取了 64 维的描述符。

至此，给定一幅含人脸的图像，我们不但可从中检测出人脸区域，而且可从这些人脸区域中提取出一些描述符。那么接下来应该做哪些工作呢？

3.6.4 Neighborhood类

至此，我们已掌握了充足的信息来构建 Neighborhood 类，以帮助我们找到最接近的特征，并记录与包含那些特征的图像关联的属性。

我们希望从一个更大的特征库（集）中匹配该图像的每个特征。只要我们能够找到与这些特征最接近的 K 个特征，也就找到了与这 K 个特征分别关联的 K 幅图像，从而也就得到了与这些图像分别关联的属性。利用这些属性，便可得到最终的分类结果。

依据我们之前掌握的关于寻找近邻的知识，可依不同的方式来解决这个问题。我们可将 Mahalanobis 距离、欧氏距离，甚至出租车距离作为距离度量。关于数据的分布，我们实在知之甚少，因此采用最简单也是最常见的距离度量——欧氏距离。这种距离函数既好用又普遍，后面的章节中我们可能还会用到它。



K-D 树或 *K* 维树是一种用于将向量与组织为树形的向量集进行匹配的数据结构。你可利用它来实现快速 *K* 近邻查找。

首先定义测试：

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  it 'finds the nearest id for a given face' do
    files = ['./test/fixtures/avatar.jpg']
    n = Neighborhood.new(files)

    n.nearest_feature_ids(files.first, 1).each do |id|
      n.file_from_id(id).must_equal files.first
    end
  end
end
```

对上述代码进行完善，可得到如下所示的 `Neighborhood` 类。

```
# lib/neighborhood.rb

class Neighborhood
  def initialize(files)
    @ids = {}
    @files = files
    setup!
  end

  def file_from_id(id)
    @ids.fetch(id)
  end

  def nearest_feature_ids(file, k)
    desc = Face.new(file).descriptors

    ids = []

    desc.each do |d|
      ids.concat(@kd_tree.find_nearest(d, k).map(&:last))
    end

    ids.uniq
  end
end
```

我们注意到，`nearest_feature_id` 函数返回的 `id` 中是没有重复元素的。这是因为我们需要从一个不存在重复元素的特征集中寻找最近邻匹配，而对待分类的特征集中是否存在重复元素并不关心。现在，既然我们已经拥有了一个可读取简单文件（即只含人脸区域的图像）的邻域，接下来需要对一些真实数据进行标注，并用它们来检验 *K* 近邻分类器的效果。为此，我们需要找到一个人脸数据库。

1. 利用人脸图像构造邻域

为使我们的胡须和眼镜检测程序能够达到期望的效果，需要一组含胡须、眼镜、既有胡须又有眼镜，以及两者均无的图像。AT&T 人脸库（http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html）是一个很好的选择。该数据集中包含了 40 个人（又称主题，subject），其中每个人都有 10 幅不同的图像，但缺少我们所需要的标注信息。因此，我们需要手工生成一些名为 attributes.json 的 JSON 文件，并将其放在每个主题对应的文件夹中。其格式如下：

```
{
  "facial_hair": false,
  "glasses": false,
}
```

如果一个主题的有些图像中含眼镜，而其他图像不含眼镜，则可使用下列格式的 attributes.json 文件：

```
[{
  "ids": [1,2,5,6,7,8,9,10],
  "facial_hair": true,
  "glasses": true,
},
{
  "ids": [3,4],
  "facial_hair": true,
  "glasses": false,
}]
```

可以看到，两个 ID 数组将属性划分为两个部分。我们还需要为将属性依附于图像这个功能编写一项测试：

```ruby
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  it 'returns attributes from given files' do
    files = ['./test/fixtures/avatar.jpg']

    n = Neighborhood.new(files)

    expected = {
      'fixtures' => JSON.parse(File.read('./test/fixtures/attributes.json'))
    }

    n.attributes.must_equal expected
  end
end
```

接着，应解析与每个文件夹对应的属性，并将其放入一个散列表：

```ruby
# lib/neighborhood.rb

class Neighborhood
  # 初始化
  # file_from_id
  # nearest_feature_ids

  def attributes
    attributes = {}
    @files.each do |file|
      att_name = File.join(File.dirname(file), 'attributes.json')

      attributes[att_name.split("/")[-2]] = JSON.parse(File.read(att_name))
    end
    attributes
  end
end
```

但我们仍然缺少一个片段，即包含了不同类别的记录（“Facial Hair No Glasses”“Facial Hair Glasses”“Glasses No Facial Hair”“Glasses Facial Hair”）的一个散列表。本例中，该散列表的形式如下：

```ruby
{
  'glasses' => {false => 1, true => 0},
  'facial_hair' => {false => 1, true => 1}
}
```

从中可以看出每类的得票数（计数）。对此进行测试的代码如下：

```ruby
# test/lib/neighborhood.rb

describe Neighborhood do
  it 'finds the nearest face which is itself' do
    files = ['./test/fixtures/avatar.jpg']
    neighborhood = Neighborhood.new(files)

    descriptor_count = Face.new(files.first).descriptors.length
    attributes = JSON.parse(File.read('./test/fixtures/attributes.json'))

    expectation = {
      'glasses' => {
        attributes.fetch('glasses') => descriptor_count,
        !attributes.fetch('glasses') => 0
      },
      'facial_hair' => {
        attributes.fetch('facial_hair') => descriptor_count,
        !attributes.fetch('facial_hair') => 0
      }
    }

    neighborhood.attributes_guess(files.first).must_equal expectation
  end
```

```
  it 'returns the proper face class' do
    file = './public/att_faces/s1/1.png'
    attrs = JSON.parse(File.read('./public/att_faces/s1/attributes.json'))

    expectation = {'glasses' => false, 'facial_hair' => false}

    attributes = %w[glasses facial_hair]
    Neighborhood.face_class(file, attributes).must_equal expectation
  end
end
```

填充完这些片段后，便得到了如下的 Neighborhood 类：

```
# lib/neighborhood.rb

class Neighborhood
  # 初始化
  # file_from_id
  # nearest_feature_ids
  # 属性

  def self.face_class(filename, subkeys)
    dir = File.dirname(filename)
    base = File.basename(filename, '.png')

    attributes_path = File.expand_path('../attributes.json', filename)
    json = JSON.parse(File.read(attributes_path))

    h = nil

    if json.is_a?(Array)
      h = json.find do |hh|
        hh.fetch('ids').include?(base.to_i)
      end or
      raise "Cannot find #{base.to_i} inside of #{json} for file #{filename}"
    else
      h = json
    end

    h.select {|k,v| subkeys.include?(k) }
  end

  def attributes_guess(file, k = 4)
    ids = nearest_feature_ids(file, k)

    votes = {
      'glasses' => {false => 0, true => 0},
      'facial_hair' => {false => 0, true => 0}
    }

    ids.each do |id|
      resp = self.class.face_class(@ids[id], %w[glasses facial_hair])

      resp.each do |k,v|
```

```
        votes[k][v] += 1
      end
    end

    votes
  end
end
```

现在我们面临的任务是如何让程序发挥作用。你可能已经注意到，`attributes_guess` 的默认值为 K。因此，我们还需要通过一些科学手段来确定 K。

2. 通过交叉验证选择*K*

现在我们需要对模型进行训练，以构建一个最优模型，并找到合适的 K 值。为此，首先将 AT&T 数据库划分为两个部分：

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  let(:files) { Dir['./public/att_faces/**/*.png'] }

  let(:file_folds) do
    {
      'fold1' => files.each_with_index.select {|f, i| i.even? }.map(&:first),
      'fold2' => files.each_with_index.select {|f, i| i.odd? }.map(&:first)
    }
  end

  let(:neighborhoods) do
    {
      'fold1' => Neighborhood.new(file_folds.fetch('fold1')),
      'fold2' => Neighborhood.new(file_folds.fetch('fold2'))
    }
  end
end
```

接下来，我们希望为每份数据构建一个测试，并通过交叉验证来查看不同的 K 所对应的误差情况。这里的测试并非指单元测试，因为我们要做的是一系列试验。我们所使用的代码如下：

```
# test/lib/neighborhood_spec.rb

describe Neighborhood do
  %w[fold1 fold2].each_with_index do |fold, i|
    other_fold = "fold#{(i + 1) % 2 + 1}"
    it "cross validates #{fold} against #{other_fold}" do
      (1..7).each do |k_exp|
        k = 2 ** k_exp - 1
        errors = 0
        successes = 0

        dist = measure_x_times(2) do
```

```
            file_folds.fetch(fold).each do |vf|
              face_class = Neighborhood.face_class(vf, %w[glasses facial_hair])
              actual = neighborhoods.fetch(other_fold).attributes_guess(vf, k)

              face_class.each do |k,v|
                if actual[k][v] > actual[k][!v]
                  successes += 1
                else
                  errors += 1
                end
              end
            end
          end

          error_rate = errors / (errors + successes).to_f

          avg_time = dist.reduce(Rational(0,1)) do |sum, bm|
            sum += bm.real * Rational(1,2)
          end
          print "#{k}, #{error_rate}, #{avg_time}\n"
        end
      end
    end
  end
```

这段代码可打印出在寻找最优 *K* 的过程中的一些有价值的信息，如图 3-16 所示（不同的 *K* 所对应的误差大小和相应的运行时间请参阅表 3-4 和表 3-5）。

表3-4：用于交叉验证的数据1

K	错误率	运行时间
1	0.0	1.428 440 5
2	0.0	0.879 999 5
4	0.057 5	1.303 254 5
8	0.177 5	2.121 337
16	0.252 5	3.758 390 5
32	0.255	8.555 531
64	0.255	23.308 074 5

表3-5：用于交叉验证的数据2

K	错误率	运行时间
1	0.0	1.477 314 5
2	0.0	0.916 875 5
4	0.05	1.309 703 5
8	0.21	2.118 357 5
16	0.247 5	3.890 095
32	0.247 5	8.624 577 5
64	0.247 5	23.480 187

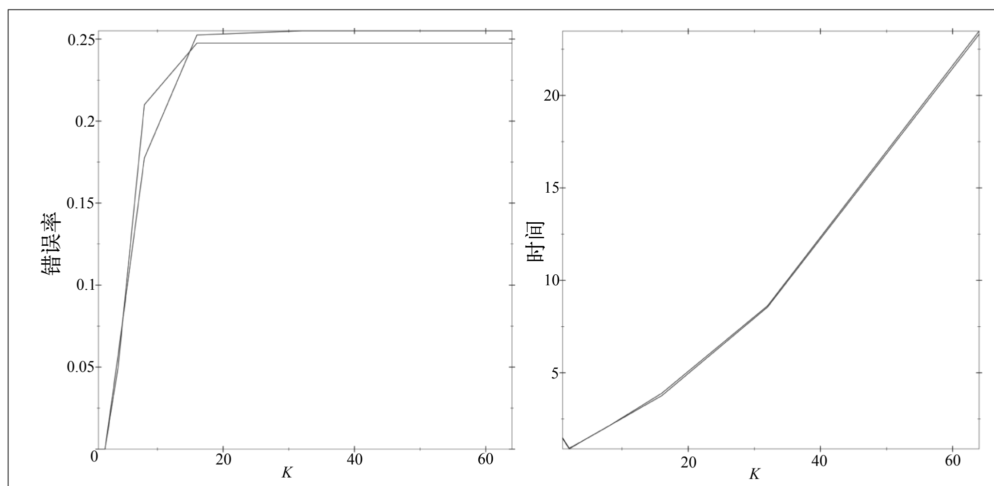


图 3-16：不同的 *K* 所对应的错误率及时间（单位为秒）

现在，对于如何选择 *K*，我们已经可以做出更好的判断了。然而，我们不应选择使错误率最小的 *K*=1 或 *K*=2，而应选择 *K*=4。这是因为当新数据到来时，我们希望对所有四个类别进行检查，从而使模型对数据集将来可能发生的变化更加稳健。

至此，我们的代码已经完全能够正常工作了！

3.7　小结

K 近邻算法是用于数据分类的最佳算法之一，它也一种懒惰的和非参数的方法。如果使用了类似 K-D 树这样的结构，则可显著提升该算法的运行效率。通过本章的学习，你还了解到在任何希望对投票进行建模或确定对象之间是否接近的场合，KNN 算法都是适用的。

你还学习了与 KNN 有关的若干重要问题，如维数灾难。当构建自己的工具以确定某人是否佩戴了眼睛或有胡须时，我们迅速地了解到，如果查看所有的像素，该任务将无从下手，因此我们借助 SURF 来实现降维。

由于 KNN 算法的渐近错误率上界由贝叶斯错误率确定，因此对于许多分类问题，首先尝试 KNN 算法来确定该问题能否解决，都不失为一个明智的选择。

第 4 章

朴素贝叶斯分类

你是否还记得几年前的电子邮件？你可能记得，当时自己的收件箱中充满了垃圾邮件，从尼日利亚王子希望典当物品到各类药物广告。我们不得不将大量时间花费在垃圾邮件的滤除上，这曾经是一个非常严重的问题。

如今，我们在垃圾邮件滤除上所花费的时间大大减少，这多亏了 Gmail 和像 SpamAssassin 这样的专业工具。借助一个被称为朴素贝叶斯分类器（Naive Bayesian Classifier）的方法，这类工具可有效阻止垃圾邮件涌入我们的收件箱。本章将探讨这个话题以及：

- 贝叶斯定理
- 何为朴素贝叶斯分类器以及称其为“朴素”的原因
- 如何利用朴素贝叶斯分类器构建垃圾邮件过滤器



我们曾经在第 2 章介绍过，朴素贝叶斯分类器是一种有监督的概率学习方法。对于数据集中输入相互独立的场合，朴素贝叶斯分类器能够展现出良好的性能。此外，它也更适合解决那些任何属性的概率均大于零的问题。

4.1 利用贝叶斯定理找出欺诈性订单

设想你在运营一家在线商店，最近你发现有很多欺诈性订单。你粗略估计这些订单所占的比例约为 10%。换言之，在 10% 的订单中，有人企图从你的商店中窃取货物。你当然希望通过减少欺诈性订单的数量来缓解这个问题，但很快就发现自己所面对的这个问题极为复杂。

假定每个月你都会收到至少 1000 份订单。如果你逐一人工排查，无疑将花费人力、物力和财力。假设判断每个订单是否存在欺诈需要至多 60 秒，而你需要为每位客服代表所支付的时薪是 15 美元，则你每年需要花费 200 小时，并支付 3000 美元的工资。

解决该问题的另一种方法是构造一个概率值，即任意一份订单有超过 50% 的几率为欺诈性订单。此时，我们希望必须进行查看的订单数量足够少。但正是这一点使情形变得复杂，因为我们唯一能确定的只是每份订单带有欺诈性的概率为 10%。给定这样的信息，我们只能注意检查所有订单，因为每份订单不带有欺诈性的可能性更大。

假设我们发现欺诈性订单通常使用礼品卡和多个促销代码。我们如何利用这些信息来确定订单是否带有欺诈性？也就是当客户使用礼品卡时，我们如何计算这份订单带有欺诈性的概率？

为回答这个问题，我们先来回顾一下条件概率。

4.1.1 条件概率

当提到某事件发生的概率时，大多数人都理解其中的含义。例如，一份订单带有欺诈性的概率为 10%。这种表述非常直观。但当某份订单使用了礼品卡时，应如何计算它带有欺诈性的概率呢？为处理更复杂的情况，我们需要条件概率这种工具，其定义如下：

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

概率符号

一般而言，$P(E)$ 表示某个事件的发生概率。这里的“事件”有很多含义，既可表示事件 A 和 B 同时发生的概率，也可表示 A 或 B 发生的概率，还可表示已知 B 已发生时 A 发生的概率。下面将介绍这些场景下如何表示概率。

$A \cap B$ 称为**与**函数，因为它表示事件 A 与 B 的交。例如，在 Ruby 语言中，交运算可这样表示：

```
a = [1,2,3]
b = [1,4,5]

a & b #=> [1]
```

$A \cup B$ 称为**或**函数，因为它同时表示了 A 和 B。例如，在 Ruby 语言中，该运算可这样表示：

```
a = [1,2,3]
b = [1,4,5]

a | b #=> [1,2,3,4,5]
```

最后，B 已发生时，A 发生的概率用 Ruby 可表示为：

```
a = [1,2,3]
b = [1,4,5]

total = 6.0

p_a_cap_b = (a & b).length / total
p_b = b.length / total

p_a_given_b = p_a_cap_b / p_b #=> 0.33
```

该定义的大意如下：当 B 发生后，A 发生的概率等于 A 和 B 同时发生的概率除以 B 发生的概率。条件概率的计算过程如图 4-1 所示。

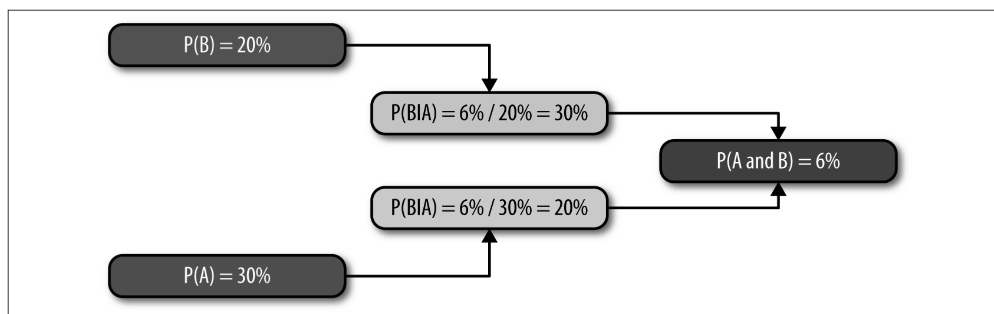


图 4-1：该图展示了$P(A-B)$与$P(A\cap B)$和$P(B)$之间的关系

在上面的欺诈性订单示例中，比如我们希望度量当某份订单使用了礼品卡时，存在欺诈的概率。由条件概率的定义可知，$P(\text{Fraud}|\text{Giftcard}) = \frac{P(\text{Fraud}\cap\text{Giftcard})}{P(\text{Giftcard})}$。这样，如果欺诈性订单、使用礼品卡的实际概率以及二者同时出现的概率已知，便可求出这个条件概率。

至此，我们再次遇到同样的问题，即由于实现难度较大，而无法计算$P(\text{Fraud}|\text{Giftcard})$。为解决这个问题，我们需要利用一个由贝叶斯发明的技巧。

4.1.2 逆条件概率

18 世纪，英国教士托马斯 · 贝叶斯（Reverend Thomas Bayes）开展了后来成为贝叶斯定理的研究。拉普拉斯（Pierre-Simon Laplace）对贝叶斯的研究进行了扩展，并得到了一个形式优美的结果，也就是我们今天所熟知的贝叶斯定理：

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

该公式之所以成立，是因为：

$$P(B|A) = \frac{\frac{P(A \cap B)P(B)}{P(B)}}{P(A)} = \frac{P(A \cap B)}{P(A)}$$

对于上面的欺诈性订单的示例，这个公式十分有用，因为我们可有效地利用其他信息来推得所要的结果。利用贝叶斯定理，我们现在便可计算：

$$P(\text{Fraud}|\text{Giftcard}) = \frac{P(\text{Giftcard}|\text{Fraud})P(\text{Fraud})}{P(\text{Giftcard})}$$

前面我们提到欺诈性订单的概率为 10%。假设使用礼品卡的概率为 10%，基于我们的研究，欺诈性订单中使用礼品卡的概率为 60%。那么，当一份订单使用了礼品卡时，它带有欺诈性的概率有多大？

$$P(\text{Fraud}|\text{Giftcard}) = \frac{60\%\dot{1}0\%}{10\%} = 60\%$$

上式的优雅之处在于统计欺诈性订单的工作量显著减少了，因为现在我们只需检查那些使用了礼品卡的订单。由于订单总数为 1000，而其中欺诈性订单的数量为 100，我们只需对其中 60 份欺诈订单进行检查便可。在其余 900 份订单中，共有 90 份使用了礼品卡，因此我们总共需要检查的订单只有 150 份！

至此，你会发现需要进行欺诈评估的订单数目从 1000 减少为 40（即总量的 4%）。是否还有改进的余地？如果人们使用了多个促销代码或其他信息，应如何应对？

4.2 朴素贝叶斯分类器

我们已经解决了从使用礼品卡的订单中找出欺诈性订单的问题，但如果这些订单中可能使用了礼品卡、多个促销代码或具有其他特征时，应当如何解决问题？

即，我们希望解决的问题是$P(A|B,C) = ?$。为此，我们还需要一些其他信息，并利用一种称为链式法则（chain rule）的工具。

4.2.1 链式法则

回顾一下概率课程，我们知道 A 和 B 均发生的概率等于 A 已发生时 B 发生的概率乘以 A 发生的概率。这个关系用数学语言可表示为$P(A \cap B) = P(B|A)P(A)$。该公式假设这些事件并不互斥。利用联合概率（joint probability），该结果可延伸出链式法则。

联合概率是指所有事件均发生的概率。若用求交符号∩来表示不同事件均发生这个事实，

则链式法则的一般形式如下：

$$P(A_1, A_2, \cdots, A_n) = P(A_1)P(A_2 \mid A_1)P(A_3 \mid A_1, A_2)\cdots P(A_n \mid A_1, A_2, \cdots, A_{n-1})$$

该扩展版本为贝叶斯概率估计注入了大量信息，从而有助于问题的求解。但这里仍然存在一个问题：它会迅速演化为涉及我们不曾拥有的信息的复杂计算，因此我们需要作出大胆的假设，并使用一些朴素的方法。

4.2.2 贝叶斯推理中的朴素性

对于解决那些潜在的相容性问题（inclusive problem），链式法则极有价值，但我们却无力计算其中涉及的所有概率。例如，如果我们打算在欺诈性订单那个示例中引入多个促销代码，则需要计算的概率为：

$$P(\text{Fraud} \mid \text{Giftcard}, \text{Promos}) = \frac{P(\text{Giftcard}, \text{Promos} \mid \text{Fraud})P(\text{Fraud})}{P(\text{Giftcard}, \text{Promos})}$$

考虑到上式中分母与订单是否带有欺诈性无关，我们暂时将其忽略。此时，我们需要将精力集中在 $P(\text{Fraud} \mid \text{Gifrcard}, \text{Promos})$ 的计算上。由链式法则可知，该式等于 $P(\text{Fraud}, \text{Gifrcard}, \text{Pr omos})$。

通过下式可证明这种关系成立：

$$P(\text{Fraud}, \text{Gift}, \text{Promo}) = P(\text{Fraud})P(\text{Gift}, \text{Promo} \mid \text{Fraud})$$
$$= P(\text{Fraud})P(\text{Gift} \mid \text{Fraud})P(\text{Promo} \mid \text{Fraud}, \text{Gift})$$

现在，我们遇到了一个棘手的问题：如何度量当一份订单带有欺诈性且使用了礼品卡时，某个促销代码的出现概率？虽然有必要计算这个概率值，但难度却很大——尤其是对于涉及更多特征的情形。如果我们从朴素的观点出发，并不关心促销代码与礼品卡之间的关系，即认为当订单带有欺诈性时，是否使用促销代码与是否使用礼品卡无关，结果会如何？

在这种假设下，上式将得到明显简化：

$$P(\text{Fraud}, \text{Gift}, \text{Promo}) = P(\text{Fraud})P(\text{Gift} \mid \text{Fraud})P(\text{Promo} \mid \text{Fraud})$$

可以看出，它与分子成正比。为进一步简化，我们可断言今后将用某个神秘的 Z（即所有类别的概率之和）将其归一化。从而，我们的模型变为：

$$P(\text{Fraud} \mid \text{Gift}, \text{Promo}) = \frac{1}{Z}P(\text{Fraud})P(\text{Gift} \mid \text{Fraud})P(\text{Promo} \mid \text{Fraud})$$

为将其转化为一个分类问题，只需确定哪个输入（带有欺诈性或不具欺诈性）的概率最大（请参阅表 4-1）。

表4-1：礼品卡与促销码的概率

	有欺诈性	无欺诈性
出现礼品卡	60%	10%
使用了多个促销代码	50%	30%
类概率	10%	90%

这时，你可利用该信息单纯地依据某份订单是否使用了礼品卡或多个促销代码来判定它是否带有欺诈性。在使用礼品卡和多个促销代码的前提下，订单带有欺诈性的概率为 62.5%。虽然就你必须核查的订单总数而言，我们无法精确指出这能帮助你节省多少工作量，但至少我们清楚我们使用了更有价值的信息，且作出了更好的决策。

尽管如此，仍然存在一个问题：如果已知一份订单带有欺诈性，而它使用多个促销代码的概率为 0，会出现什么情况？造成这种结果的原因有多种，其中一种是样本容量不足。伪计数便是应对这种情况的一种方法。

4.2.3 伪计数

对于朴素贝叶斯分类器而言，有一个巨大的挑战，即新信息的引入。例如，我们有一大宗电子邮件，分类为普通邮件或垃圾邮件。我们利用所有这些数据构建了一些概率值，但接着出现了一种糟糕的情形：一个新单词“fuzzbolt”出现了。由于这个词在我们的数据中未曾出现，因此通过计算得到给定该单词，邮件为垃圾邮件的概率为 0。这会导致归零效应（zero-out effect），使得结果向我们拥有的数据产生显著的偏倚。

由于为得到分类结果，朴素贝叶斯分类器需要将所有条件独立的概率相乘，因此一旦这些概率之中有一项为 0，则总概率也将为 0。

例如，假设现有一封标题为“Fuzzbolt: Prince of Nigeria”的邮件。假设我们将单词“of”移除，从而得到如表 4-2 所示的概率值。

表4-2：给定垃圾邮件或普通邮件时单词的概率

单词	垃圾邮件	普通邮件
Fuzzbolt	0	0
Prince	75%	15%
Nigeria	85%	10%

现在假设我们希望计算出这封邮件“是垃圾邮件的得分”和“是普通邮件的得分”两项分值。在这两种情形中，由于 fuzzbolt 并未出现，因此分值都将为 0。这时，由于出现了得

分相等的情况，我们依据从众原则，将这封邮件判定为普通邮件。这意味着我们预测失败，由于一个未被识别的单词，得到了错误的分类结果。

对于该问题有一种简单的修正方法——伪计数（pseudocount）。在计算概率时，我们为单词的出现次数加上 1，即所有单词的出现次数均记为 `word_count+1`。这有助于缓解归零效应的影响。在上面的欺诈性订单检测示例中，通过为每个单词的出现次数增加 1，便可确保概率值中永远不会出现零值。

假设在上面的例子中，共有 3000 个单词。我们给 fuzzbolt 的评分是 1/3000。其他分数虽然产生了细微的变化，但却规避了归零效应。

4.3 垃圾邮件过滤器

构建垃圾邮件过滤器是典型的机器学习示例。本节中，我们将使用朴素贝叶斯分类器搭建一个简单的垃圾邮件过滤器，并利用三元符号化模型（3-gram tokenization model）改进其性能。

通过前面的学习，我们已了解到朴素贝叶斯分类器非常易于计算，且在强条件独立假设满足时，具有优异的性能。本例中，我们将介绍以下内容：

- 彼此交互的类如何定义
- 一个高质量的数据源
- 一种符号化模型
- 一个用于将误差最小化的目标函数
- 一种随时间提升性能的方法



安装说明

本例所使用的全部代码都可从 GitHub 站点 https://github.com/thoughtfulml/examples/tree/master/3-naive-bayesian-classification 免费下载。

由于 Ruby 语言在不断地更新，为保证代码的正确运行，最好参考相关的 README 文件。

运行代码前，请确保事先安装了 `libxml` 库。

4.3.1 类图

本例中，每封电子邮件都对应于一个可接收 `.eml` 类型的文本文件（即新邮件），并可将其符号化为 `SpamTrainer` 类可利用的信息的对象。完整的类图请参阅图 4-2。

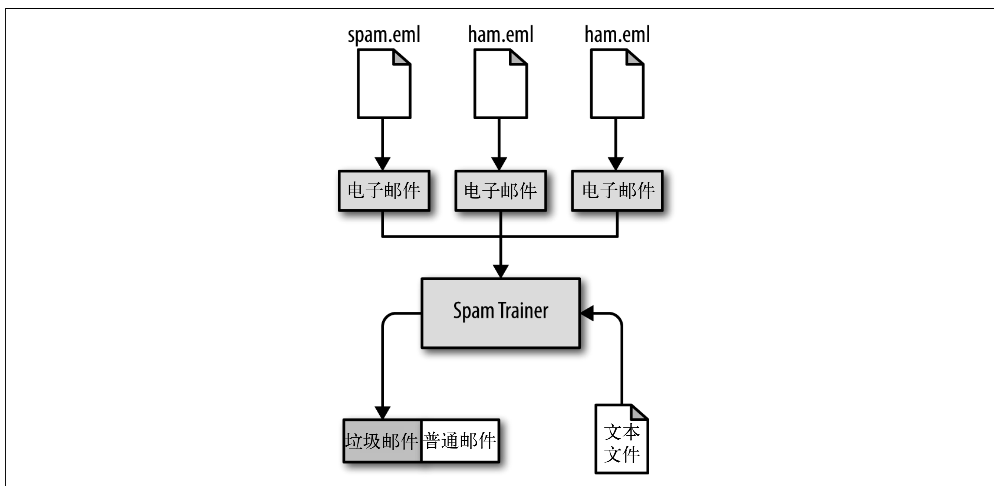


图 4-2：展示了电子邮件如何输入 SpamTrainer 类的类图

4.3.2 数据源

有大量数据源可供选择，但最适合我们的是带有是否为垃圾邮件标注信息的数据集。为此，可利用 CSDMC2010 SPAM 数据集，它可从 SourceForge 站点 http://csmining.org/index.php/spam-email-datasets-.html 免费获取。

该数据集共含 4327 封邮件，其中普通邮件 2949 封，垃圾邮件 1378 封。这个数据集的规模不算很大，但对于验证概念，已经足够了。

4.3.3 Email类

Email 类的功能较单一，只负责依据邮件的 RFC 对新收到的电子邮件进行解析。为此，我们使用 gem 包，因为其邮件处理功能非常强大。但在我们的模型中，只关心标题和邮件正文。

我们需要处理的信息类型包括 HTML 消息、纯文本（plaintext）、多部分（multipart）等，而其他类型的信息将忽略。

下面我们利用测试驱动开发一步步构建该类。

从简单的纯文本开始，我们将数据集中的一个训练样本文件从 data/TRAINING/TRAIN_00001.eml 复制到 ./test/fixtures/plain.eml。该文件是一个纯文本文件，很适合我们。请注意，电子邮件中正文和标头用 \r\n\r\n 分隔。与标头信息一起的通常是一些如“标题：此处为邮件标题”之类的文字。利用这一点，我们可轻松提取出测试样例：

```
require 'spec_helper'

# test/lib/email_spec.rb

describe Email do
  describe 'plaintext' do
    let(:plain_file) { './test/fixtures/plain.eml' }
    let(:plaintext) { File.read(plain_file) }
    let(:plain_email) { Email.new(plain_file) }

    it 'parses and stores the plain body' do
      body = plaintext.split("\n\n")[1..-1].join("\n\n")
      plain_email.body.must_equal body
    end

    it 'parses the subject' do
      subject = plaintext.match(/^Subject: (.*)$/)[1]
      plain_email.subject.must_equal subject
    end
  end
end
```

现在，我们并不打算单纯地依赖正则表达式，而是希望利用可处理这些细节的邮件包，具体实现如下：

```
require 'forwardable'

# lib/email.rb

class Email
  extend Forwardable

  def_delegators :@mail, :subject

  def initialize(filepath)
    @filepath = filepath
    @mail = Mail.read(filepath)
  end

  def body
    @mail.body.decoded
  end
end
```

你会注意到，我们利用了 `def_delegators` 将邮件标题委托给 `@mail` 对象。这样做只是为了简单起见。

既然我们已经能够读取纯文本邮件，接下来考虑如何读取 HTML 邮件。为此，我们希望仅提取内部文字 `inner_text`。由于正则表达式在这种场合无法发挥作用，我们需要借助另外一个包——Nokogiri，它能够大大简化我们的工作。但我们首先需要一个如下所示的测试用例：

```ruby
# test/lib/email_spec.rb

describe Email do
  describe 'html' do
    let(:html_file) { './test/fixtures/html.eml' }
    let(:html) { File.read(html_file) }
    let(:html_email) { Email.new(html_file) }

    it "parses and stores the html body's inner_text" do
      body = html.split("\n\n")[1..-1].join("\n\n")
      html_email.body.must_equal Nokogiri::HTML.parse(body).inner_text
    end

    it "stores subject like plaintext does as well" do
      subject = html.match(/^Subject: (.*)$/)[1]
      html_email.subject.must_equal subject
    end
  end
end
```

如前所述，我们准备利用 Nokogiri 来计算 `inner_text`，而且是在 `Email` 类的内部使用它。现在的问题在于我们还需要检测 `content_type`。因此我们将它也加入 `Email` 类的定义：

```ruby
# lib/email.rb

require 'forwardable'

class Email
  extend Forwardable

  def_delegators :@mail, :subject, :content_type

  def initialize(filepath)
    @filepath = filepath
    @mail = Mail.read(filepath)
  end

  def body
    single_body(@mail.body.decoded, content_type)
  end

  private
  def single_body(body, content_type)
    case content_type
    when 'text/html'
      Nokogiri::HTML.parse(body).inner_text
    when 'text/plain'
      body.to_s
    else
      ''
    end
  end
end
```

此时，我们也可添加多部分处理，但我希望将它留作练习。你可从本书的配套代码中找到多部分处理的版本。

这样，我们就拥有了一个可以工作的电子邮件解析程序，但我们还需要处理符号化，或从邮件正文和标题中提取单词。

4.3.4 符号化与上下文

如图 4-3 所示，符号化文本有多种方法，如依据词干、词频和单词进行符号化。对于垃圾邮件，我们遇到了一个由上下文引起的非常有挑战性的问题。短语 Buy now 听起来很像垃圾邮件，而 Buy 和 now 则没有这种感觉。由于我们要构建的是一个朴素贝叶斯分类器，因此假设每个独立的符号（token）对邮件是否为垃圾邮件都有贡献。

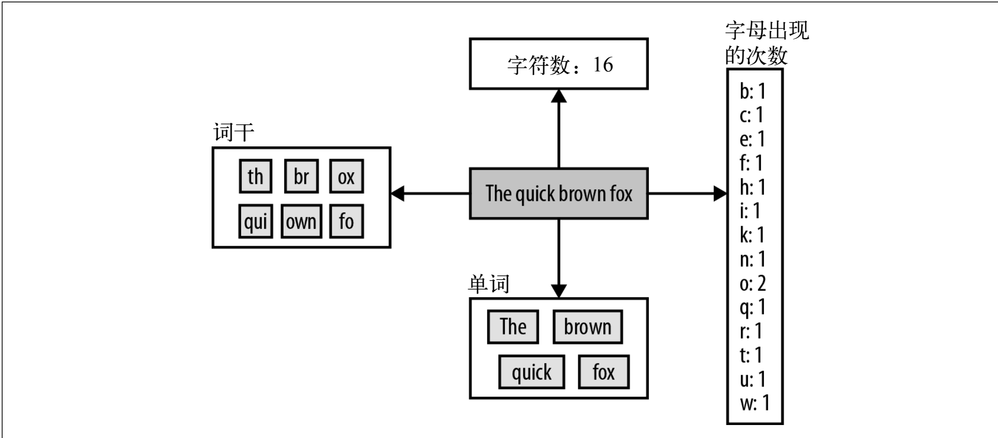


图 4-3：符号化文本有多种方法

我们所要构建的符号化程序的目标是将单词提取为流。我们并不打算返回一个数组，而是希望每个符号出现时才生成它，这样程序运行期间便可保持较低的内存占用率。为保持一致，我们事先将所有字符串都转化为小写：

```
# test/lib/tokenizer_spec.rb
require 'spec_helper'

describe Tokenizer do
  describe '1-gram tokenization' do
    it 'downcases all words' do
      expectation = %w[this is all caps]
      Tokenizer.tokenize("THIS IS ALL CAPS") do |token|
        token.must_equal expectation.shift
      end
    end

    it 'uses the block if given' do
```

```
      expectation = %w[feep foop]
      Tokenizer.tokenize("feep foop") do |token|
        token.must_equal expectation.shift
      end
    end
  end
end
```

之前曾交待过，利用这段符号化代码，我们要完成两项任务。首先，将所有单词中的字母都转换为小写；其次，我们使用了块，而非数组。这样做是为了降低内存消耗，因为我们无需构建数组并将其返回。这使得它更加“懒惰”。为了完成后续测试工作，我们需要往符号化模块的框架中填充一些代码，如下所示：

```
# lib/tokenizer.rb

module Tokenizer
  extend self

  def tokenize(string, &block)
    current_word = ''
    return unless string.respond_to?(:scan)
    string.scan(/[a-zA-Z0-9_\u0000]+/).each do |token|
      yield token.downcase
    end
  end
end
```

既然我们已经掌握了邮件的解析和符号化方法，接下来便可进入贝叶斯部分——SpamTrainer。

4.3.5 SpamTrainer类

现在我们需要构建 SpamTrainer 类，以利用它来完成三项任务。

(1) 存储训练数据。
(2) 构建贝叶斯分类器。
(3) 通过测试将假正率最小化。

1. 训练数据的存储

我们要做的第一件事是从给定的邮件消息集中提取和存储训练数据。在产品环境中，你会选择一些具有持久性的结构。本例中，我们将一切都保存在一个很大的散列中。

请牢记，大多数机器学习算法都有两个步骤：训练和计算。我们的训练步骤又由以下三个子步骤组成。

(1) 存储所有的类别。
(2) 存储每类中每个单词（需事先进行去重处理）的出现频数。
(3) 存储每类中所有单词的出现频数之和。

因此，我们需要首先获取所有类别的名称；测试的内容大致如下：

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  describe 'initialization' do
    let(:hash_test) do
      {'spam' => './filepath', 'ham' => './another', 'scram' => './another2'}
    end

    it 'allows you to pass in multiple categories' do
      st = SpamTrainer.new(hash_test)
      st.categories.sort.must_equal hash_test.keys.uniq.sort
    end
  end
end
```

解决方案如下列代码所示：

```
# lib/spam_trainer.rb

class SpamTrainer
  def initialize(training_files, n = 1)
    @categories = Set.new

    training_files.each do |tf|
      @categories << tf.first
    end
  end
end
```

你会注意到，我们目前利用了一个集合来保存类的信息，因为集合的性质能够保证各类别不会重复出现。接下来是从每封邮件中提取无重复的符号。我们利用一个特殊的类别 `_all` 来记录所有符号的出现频数：

```
Subject: One of a kind Money maker! Try it for free!

spam

# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  let(:training) do
    [['spam','./test/fixtures/plain.eml'], ['ham','./test/fixtures/small.eml']]
  end

    let(:trainer) { SpamTrainer.new(training)}
```

```
  it 'initializes counts all at 0 plus an _all category' do
    st = SpamTrainer.new(hash_test)
    %w[_all spam ham scram].each do |cat|
      st.total_for(cat).must_equal 0
    end
  end
end
```

为了使之工作，我们需要引入一个名为 train! 的新方法。该方法应能够接收训练数据，对其遍历，并将数据保存到一个内部散列中。解决方案如下：

```
# lib/spam_trainer.rb

class SpamTrainer
  def initialize(training_files)
    setup!(training_files)
  end

  def setup!(training_files)
    @categories = Set.new

    training_files.each do |tf|
      @categories << tf.first
    end

    @totals = Hash[@categories.map {|c| [c, 0]}]
    @totals.default = 0
    @totals['_all'] = 0

    @training = Hash[@categories.map {|c| [c, Hash.new(0)]}]
  end

  def total_for(category)
    @totals.fetch(category)
  end

  def train!
    @to_train.each do |category, file|
      write(category, file)
    end
    @to_train = []
  end

  def write(category, file)
    email = Email.new(file)

    logger.debug("#{category} #{file}")

    @categories << category
    @training[category] ||= Hash.new(0)

    Tokenizer.unique_tokenizer(email.blob) do |token|
      @training[category][token] += 1
```

```
      @totals['_all'] += 1
      @totals[category] += 1
    end
  end
end
```

现在我们已经在程序中的训练部分投入了不少精力，但对于其性能却一无所知。而且，它还不能完成任何分类任务。为此，我们还需构建一个分类器。

2. 构建贝叶斯分类器

我们首先回忆一下贝叶斯公式：

$$P(A_i \mid B) = \frac{P(B \mid A_i)P(A_i)}{\sum_j P(B \mid A_j)P(A_j)}$$

由于我们对贝叶斯公式的认识还很浅，所以将它表示为更简单的形式：

$$\text{Score}(\text{Spam}, W_1, W_2, \cdots, W_n) = P(\text{Spam})P(W_1 \mid \text{Spam})P(W_2 \mid \text{Spam})\cdots P(W_n \mid \text{Spam})$$

之后我们用某个归一化常量 Z 去除上式。

我们接下来的目标是构建 `score` 方法、`normalized_score` 方法和 `classify` 方法。`score` 方法是从之前的计算得到的原始分值，而 `normalized_socre` 的取值范围为 0~1（通过除以所有符号的出现频数总和 Z 实现归一化）。

`score` 方法的测试代码如下：

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  describe 'scoring and classification' do
    let (:training) do
      [
        ['spam','./test/fixtures/plain.eml'],
        ['ham','./test/fixtures/plain.eml'],
        ['scram','./test/fixtures/plain.eml']
      ]
    end

    let(:trainer) do
      SpamTrainer.new(training)
    end

    let(:email) { Email.new('./test/fixtures/plain.eml') }

    it 'calculates the probability to be 1/n' do
      scores = trainer.score(email).values

      assert_in_delta scores.first, scores.last
```

```
      scores.each_slice(2) do |slice|
        assert_in_delta slice.first, slice.last
      end
    end
  end
end
```

由于在所有类别上训练数据都是均匀分布的，因此没有理由为它们分配不同的分值。为此，我们需要为 SpamTrainer 类的定义添加下列代码片段：

```
# lib/spam_trainer.rb

class SpamTrainer
  #def initialize
  #def total_for
  #def train!
  #def write

  def score(email)
    train!

    unless email.respond_to?(:blob)
      raise 'Must implement #blob on given object'
    end

    cat_totals = totals

    aggregates = Hash[categories.map do |cat|
      [
        cat,
        Rational(cat_totals.fetch(cat).to_i, cat_totals.fetch("_all").to_i)
      ]
    end]

    Tokenizer.unique_tokenizer(email.blob) do |token|
      categories.each do |cat|
        r = Rational(get(cat, token) + 1, cat_totals.fetch(cat).to_i + 1)
        aggregates[cat] *= r
      end
    end

    aggregates
  end
end
```

该测试的内容如下。

- 若模型尚未训练（train! 方法负责训练）好，则训练模型。
- 对于当前邮件正文中的每个符号，遍历所有类别，并计算当前符号来自每个类别的概率，从而计算出未经 Z 归一化的朴素贝叶斯分值。

在获得每个符号的分值之后，还需定义方法 normalized_score，确保这些分值之和为 1。

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  it 'calculates the probability to be exactly the same and add up to 1' do
    trainer.normalized_score(email).values.inject(&:+).must_equal 1
    trainer.normalized_score(email).values.first.must_equal Rational(1,3)
  end
end
```

这样，SpamTrainer 类最终为：

```
# lib/spam_trainer.rb

class SpamTrainer
  #def initialize
  #def total_for
  #def train!
  #def write
  #def score

  def normalized_score(email)
    score = score(email)
    sum = score.values.inject(&:+)

    Hash[score.map do |cat, aggregate|
      [cat, (aggregate / sum).to_f]
    end]
  end
end
```

3. 分类的计算

得到符号的分值之后，为便于最终用户使用，我们还需计算分类结果。分类也以对象的形式定义，并应返回预测结果和分值。但在预测结果中，可能出现预测分值并列的问题。

例如，假设现有一个模型，涉及火鸡和豆腐两个类别。如果对两类的预测分值旗鼓相当，该如何处理？可能最佳方式是看哪个类别的出现概率更大，然后将结果判为该类。但如果两类的类概率仍然相同，该如何处理？此时，我们可依字母次序来决定最终的预测结果。

对此进行测试时，我们需要依据各类的出现概率为它们引入一个优先级次序。测试如下：

```
# test/lib/spam_trainer_spec.rb

describe SpamTrainer do
  describe 'scoring and classification' do
    it 'sets the preference based on how many times a category shows up' do
      expected = trainer.categories.sort_by {|cat| trainer.total_for(cat) }

      trainer.preference.must_equal expected
    end
```

```
  end
end
```

测试具体实施的代码如下：

```
# lib/spam_trainer.rb

class SpamTrainer
  #def initialize
  #def total_for
  #def train!
  #def write
  #def score
  #def normalized_score

  def preference
    categories.sort_by {|cat| total_for(cat) }
  end
end
```

设置好各类别优先级后，便可测试分类结果是否正确，相应的代码如下：

```
# test/lib/spam_trainer.rb

describe SpamTrainer do
  describe 'scoring and classification' do
    it 'gives preference to whatever has the most in it' do
      score = trainer.score(email)
      preference = trainer.preference.last
      preference_score = score.fetch(preference)

      expected = SpamTrainer::Classification.new(preference, preference_score)

      trainer.classify(email).must_equal expected
    end
  end
end
```

测试具体实施的代码同样十分简单，如下所示：

```
# lib/spam_trainer.rb

class SpamTrainer
  Classification = Struct.new(:guess, :score)
  #def initialize
  #def total_for
  #def train!
  #def write
  #def score
  #def preference
  #def normalized_score

  def classify(email)
```

```
      score = score(email)
      max_score = 0.0
      max_key = preference.last
      score.each do |k,v|
        if v > max_score
          max_key = k
          max_score = v
        elsif v == max_score && preference.index(k) > preference.index(max_key)
          max_key = k
          max_score = v
        else
          # Do nothing
        end
      end
      throw 'error' if max_key.nil?
      Classification.new(max_key, max_score)
    end
  end
```

4.3.6 通过交叉验证将错误率最小化

现在我们需要对所构建的模型的性能进行评价。为此，需要利用之前下载的数据，并通过交叉验证来测试。在交叉验证中，我们只需关注假正率，并据此来决定是否需要对模型参数进行微调。

1. 最小化假正率

在此之前，我们的目标一直都是将错误率最小化。错误率可由被误分类的样本数目除以被分类的样本总数得到。在大多数情形下，这都完全符合我们的期望，但对于垃圾邮件过滤器，这并不是我们的优化目标。我们真正希望最小化的是假正例的数量。假正例也称为第一类错误，它标识模型错误地将一个负例预测为正类。

在本例中，如果某封邮件是普通邮件，却被我们的模型预测为垃圾邮件，则用户将丢失这封邮件。我们自然希望自己的垃圾邮件过滤器的假正率尽可能地低。另一方面，如果某封邮件是垃圾邮件，但被模型错误地预测为普通邮件，则我们不会特别在意这种结果。

我们并不追求总错误率（即误分类样本总数与参与分类的样本总数之比）最小化，而是希望假正率最小。我们也会度量假负率，但这个指标的重要性要次之，因为我们的目标是降低垃圾邮件进入收件箱的几率，而非完全消除垃圾邮件。

为此，首先需要从我们的数据集中获取一些信息，下一小节将对此进行介绍。

2. 构建交叉验证的数据份

在垃圾邮件训练数据内部，是一个名为 keyfile.label 的文件，其中记录了每个文件是否为垃圾邮件。在交叉验证测试中，通过下列代码，很容易对文件进行解析：

```ruby
# test/cross_validation_spec.rb

describe 'Cross Validation' do
  def self.parse_emails(keyfile)
    emails = []
    File.open(keyfile, 'rb').each_line do |line|
      label, file = line.split(/\s+/)
      emails << Email.new(filepath, label)
    end
    emails
  end

  def self.label_to_training_data(fold_file)
    training_data = []
    st = SpamTrainer.new([])

    File.open(fold_file, 'rb').each_line do |line|
      label, file = line.split(/\s+/)
      st.write(label, file)
    end

    st
  end

  def self.validate(trainer, set_of_emails)
    correct = 0
    false_positives = 0.0
    false_negatives = 0.0
    confidence = 0.0

    set_of_emails.each do |email|
      classification = trainer.classify(email)
      confidence += classification.score
      if classification.guess == 'spam' && email.category == 'ham'
        false_positives += 1
      elsif classification.guess == 'ham' && email.category == 'spam'
        false_negatives += 1
      else
        correct += 1
      end
    end

    total = false_positives + false_negatives + correct

    message = <<-EOL
    False Positives: #{false_positives / total}
    False Negatives: #{false_negatives / total}
    Accuracy: #{(false_positives + false_negatives) / total}
    EOL
    message
  end
end
```

3. 交叉验证与误差度量

从这里开始，我们可真正开始构建交叉验证测试，

```ruby
# test/cross_validation_spec.rb
describe 'Cross Validation' do
  describe "Fold1 unigram model" do
    let(:trainer) {
      self.class.label_to_training_data('./test/fixtures/fold1.label')
    }

    let(:emails) {
      self.class.parse_emails('./test/fixtures/fold2.label')
    }

    it "validates fold1 against fold2 with a unigram model" do
      skip(self.class.validate(trainer, emails))
    end
  end

  describe "Fold2 unigram model" do
    let(:trainer) {
      self.class.label_to_training_data('./test/fixtures/fold2.label')
    }

    let(:emails) {
      self.class.parse_emails('./test/fixtures/fold1.label')
    }

    it "validates fold2 against fold1 with a unigram model" do
      skip(self.class.validate(trainer, emails))
    end
  end
end
```

运行命令 `ruby test/cross_validation_spec.rb` 时，会得到下列结果：

```
WARNING: Could not parse (and so ignoring) 'From spamassassin-devel-admin@lists.
sourceforge.net Fri Oct 4 11:07:38 2002'
Parsing emails for ./test/fixtures/fold2.label
WARNING: Could not parse (and so ignoring) 'From quinlan@pathname.com Thu Oct 1
0 12:29:12 2002'
Done parsing emails for ./test/fixtures/fold2.label
Cross Validation::Fold1 unigram model
  validates fold1 against fold2 with a unigram model

        False Positive Rate (Bad): 0.0036985668053629217
        False Negative Rate (not so bad): 0.16458622283865001
        Error Rate: 0.16828478964401294

WARNING: Could not parse (and so ignoring) 'From quinlan@pathname.com Thu Oct 1
0 12:29:12 2002'
Parsing emails for ./test/fixtures/fold1.label
WARNING: Could not parse (and so ignoring) 'From spamassassin-devel-admin@lists.
```

```
sourceforge.net Fri Oct 4 11:07:38 2002'
Done parsing emails for ./test/fixtures/fold1.label
Cross Validation::Fold2 unigram model
  validates fold2 against fold1 with a unigram model

        False Positive Rate (Bad): 0.0055452865064695O1
        False Negative Rate (not so bad): 0.17375231053604437
        Error Rate: 0.17929759704251386
```

你会注意到假负率（将垃圾邮件预测为普通邮件的比例）远高于假正率（将普通邮件预测为垃圾邮件的比例），造成这种结果的根本原因在于贝叶斯定理。下面通过表 4-3 观察一下普通邮件和垃圾邮件的实际概率。

表4-3：垃圾邮件与普通邮件

类别	邮件数目	单词数目	邮件的概率	单词的概率
垃圾邮件	1378	231 472	31.8%	36.3%
普通邮件	2949	406 984	68.2%	63.7%
总计	4327	638 456	100%	100%

如你所见，普通邮件的出现概率更高，因此我们会将普通邮件作为默认类别，并常常会将疑似垃圾邮件的邮件判为普通邮件。然而，这样做的一个好处是可减少 80% 的垃圾邮件，而不会影响新到来的邮件。

4.4 小结

本章深入探讨了如何构建和理解朴素贝叶斯分类器。我们了解了该算法适用于数据相互独立的场景。作为一个概率模型，在需要将数据依据得分分类到多个分支的场合中，该方法往往具有良好的性能。这种有监督学习方法对于欺诈检测、垃圾邮件过滤以及其他拥有这些特征类型的问题都很适用。

第5章

隐马尔可夫模型

我们的许多行为都是靠直觉驱动的。例如，直觉会告诉我们某些单词可能为某种词性，或者如果一位用户访问了注册页，则他便有很大的概率成为一名客户。但我们如何去构建一个关于直觉的模型呢？

本章的主题是隐马尔可夫模型，它非常善于利用观测量以及对系统状态工作原理的假设，找到给定系统的隐含状态。在本章中，我们将首先讨论如何在给定用户行为时追踪用户的状态，然后详细讨论何为隐马尔可夫模型，最后利用布朗语料库（Brown Corpus）来构建一个词性标注器。



隐马尔可夫模型既可是有监督的，也可是无监督的。由于它们都是以马尔可夫模型为基础的，因此也称其具有无后效性（或马尔可夫性，Markovian）的。这些模型无需内置大量历史信息，便可表现出优异的性能。此外，对于需要为分类任务添加局部上下文的场合，该方法也十分适用。

5.1 利用状态机跟踪用户行为

你是否听说过销售漏斗（sales funnel）？这个概念描述的是不同层次的客户之间的转换关系。人们最初是潜在客户，而后发展为参与度更高的状态（参见图 5-1）。

假设我们拥有一个网店，并已知在访问该网站的潜在客户中有 15% 会进行注册，而 5% 会立即成为客户。若访问者已成为用户，则他会在 5% 的时间内销账，在 15% 的时间内购买商品。若访问者已成为客户，则他仅会在 2% 的时间内销账，在 95% 的时间内作为普通客

户，而不会持续地购买商品。

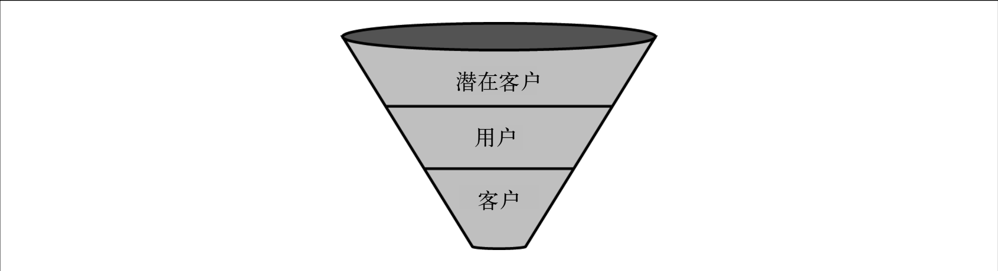


图 5-1：描述了从潜在客户到实际客户的销售漏斗



所谓“潜在客户”是指那些访问站点 1~2 次，但通常没有实际购买行为的“潜水者”。用户则倾向于浏览网站，并偶尔有购物行为。最后，客户的参与度较高，有购物行为，但通常不会在短时间内购买大量商品，因此会临时转变为用户。

可将我们所收集的信息用*转移矩阵*（transition matrix）来表示。借助它，可清晰地展示从一个状态到另一个状态的转移概率。

表5-1：转移概率

	潜在客户	用户	客户
潜在客户	0.80	0.15	0.05
用户	0.05	0.80	0.15
客户	0.02	0.95	0.03

转移概率实际上定义了一个*状态机*（如图 5-2 所示）。此外，它还揭示了关于当前客户行为模式的大量信息。我们可确定转移率、流失率和其他概率。*转移率*（conversion rate）是一名潜在客户注册的概率，即 20%。它等于从潜在客户变为用户的概率与从潜在客户变为客户的概率之和（15%+5%）。*流失率*（attrition rate）则可通过取 5% 和 2% 的均值来得到，即 3.5%。

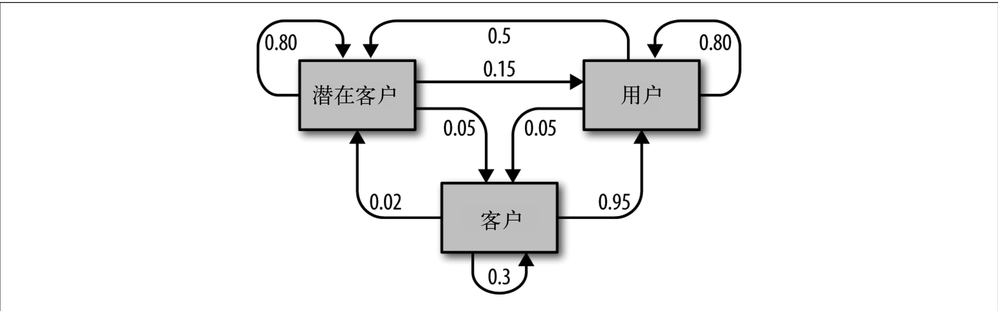


图 5-2：带有转移概率的销售漏斗状态机

在分析中，这种展示用户行为的方式并不常见，因为它过于直白。但与传统的转移率计算相比，它有一个优点，即通过它可观测用户行为随时间的变化。例如，假设某位客户之前四次均为潜在客户，则我们可确定他当前为潜在客户的概率。该值等于为潜在客户的概率（比如 80%）乘以之前四次为潜在客户的概率（均为 80%）。某人持续浏览网站但并不注册的几率很低，因为他最终可能会注册。

但这个模型的主要问题是，如果不单独询问每个用户，我们是无法可靠地确定这些状态的。也就是说用户的状态是不可观测的，因为用户可以匿名方式浏站点。

很快你将了解到，这其实问题不大。只要我们能够观察到用户与站点之间的交互，并能判断来自其他源的内部转移情况（如借助 Google Analytics），则我们仍然能够求解该问题。

为此，我们需要引入另一个层次的复杂性——*输出*（emission）。

5.1.1 隐含状态的输出和观测

在前面的例子中，我们并不清楚某人何时会从潜在客户转变为用户，或转变为客户。但我们能够观察到用户所做的事以及他的行为。我们知道，对于一个给定的观测结果，都存在一个他处于某个给定状态的概率。

可通过观察用户发出的行为来确定其隐含状态。例如，假设我们的站点中共含有 5 个页面：主页（Home）、注册页（Signup）、产品页（Product）、结账页（Checkout）以及联系页（Contact Us）。你会想到，这些页面对我们的重要性不尽相同。例如，注册页很大程度上意味着潜在客户变成了用户，而结账页则表明用户已转变为客户。

由于处在这些状态的概率已知，这些信息变得更加有趣了。假设我们已知的输出概率和状态概率如表 5-2 所示。

表5-2：输出概率和状态概率

页面名称	潜在客户	用户	客户
Home（主页）	0.4	0.3	0.3
Signup（注册页）	0.1	0.8	0.1
Product（产品页）	0.1	0.3	0.6
Checkout（结账页）	0	0.1	0.0
Contact Us（联系页）	0.7	0.1	0.2

我们已经知道了用户切换状态的概率，以及给定隐含状态他们发出某种行为的概率。我们关心的是，给定这些信息后，查看了主页、注册页、产品页的用户成为客户的概率是多少？即，我们希望解决如图 5-3 所描述的问题。

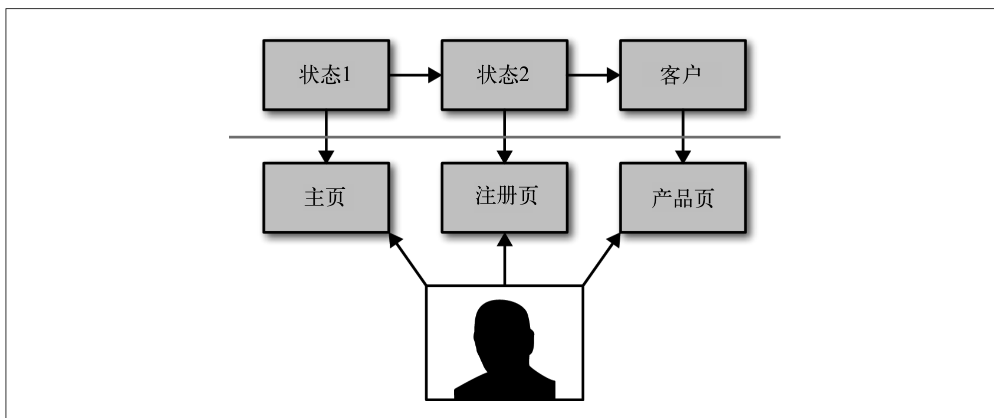


图 5-3：你仅能观察到用户所做的事，但隐含状态不可见

为解决这个问题，我们需要确定在给定用户之前的全部状态的条件下，用户处于客户状态的概率，即 $P(\text{Customer}|S_1,S_2$，以及在给定用户为客户状态的条件下，用户查看产品页面的概率与给定隐含状态 S_1、S_2 条件下分别观看主页和注册页的概率之积，即 $P(\text{Product_Page}|\text{Customer}) * P(\text{Signup_Page}|S_2) * P(\text{Homepage}|S_1)$。现在的问题在于，未知量的个数要多于已知量。

这个有限模型求解起来十分困难，因为其中涉及了大量计算。计算如 $P(\text{Customer}|S_1, S_2, \cdots, S_N)$ 这样的问题非常复杂。为解决该问题，我们需要引入马尔可夫假设。



在隐马尔可夫模型的术语中，输出（emission）和观测（observation）经常交替使用。二者实际上是等同的，均是指某个过程产生的输出或你可观测到的量。

5.1.2　利用马尔可夫假设简化问题

你可能还记得，在朴素贝叶斯分类中，每个属性会对一些事件的概率独立地产生贡献。因此，对于垃圾邮件，给定如 Prince 和 Buy now 这样的单词或短语时，各概率条件独立。而在上面的依据用户行为所构建的模型中，随机变量之间的依赖关系是我们所希望有的。具体说来，我们希望前一次的状态对下一次状态的概率产生影响。实际上，我们可以断言上一次状态与用户的当前状态存在某种关系。

在朴素贝叶斯分类模型中，我们所做的假设是，给定一些事件，某些事物的概率条件独立。因此，垃圾邮件对于邮件中的每个单词都是条件独立的。

对于我们的当前系统，也可做出同样的假设。我们可假设用户处于某个特定状态的概率主

要取决于前一个状态。因此，可将$P(\text{Customer}|S_1,S_2,\cdots,S_n)$简化为$P(\text{Customer}|S_n)$。但这种简化的依据何在？

给定如前所定义的状态机，系统可以递归地对你过去所处的状态进行概率推断。例如，如果站点的某个访问者处于客户状态，则你可以说他之前最可能的状态是用户，而在此之前他最有可能的状态是潜在客户。

从这个假设还可得出一个令人兴奋的结论，这将我们引向下一个话题——*马尔可夫链*（Markov chain）。

5.1.3 利用马尔可夫链而非有限状态机

到目前为止，我们只讨论了一个系统，而这个系统仅包含一个输出。马尔可夫假设的强大之处在于，我们可将待建模的系统视为处于永恒的运转之中。我们并不关心该过程的某个局部时间点的状态，而是希望推断该系统的稳态行为。马尔可夫链的形成正是基于这种思想。

马尔可夫链在系统模拟方面表现十分突出，它在队列理论、金融、气象建模以及博弈论中均被大量运用。这类模型之所以强大，是因为它们能够以简洁的方式表示各种行为。此外，当给定某个马尔可夫链时，我们还可迅速地确定如何对系统的行为作出假设。

借助马尔可夫链，我们可对一个永恒持续的潜在过程进行分析，并发现一些有用信息。但这并不足以解决我们的基本问题，即我们仍然需要确定当给定某人之前的隐含状态和我们自己的观察时，他的当前状态。为此，我们需要引入隐含状态来增强马尔可夫链。

5.1.4 隐马尔可夫模型

前面通过大量篇幅来讨论观测和潜在状态的转移，但我们发现问题又回到了原点。我们仍然需要指出用户的当前状态。为此，我们可使用*隐马尔可夫模型*（Hidden Markov Model）。该模型由下列三个要素构成。

- *评估*：如“主页→注册页→产品页→结账页”这样的序列在我们掌握的用户状态转移和观测中的出现概率有多大？
- *解码*：给定该序列，最可能的隐状态序列是什么？
- *学习*：给定观测序列，用户接下来最有可能的行为是什么？

在接下来的几节中，我们将详细讨论这三个要素。首先，我们将介绍用于评估观测序列的前向–后向算法（Forward-Backward algorithm）；之后，我们将深入探讨如何利用维特比算法来解决解码问题；最后，作为对解码的扩展，我们将介绍学习的主要思想。

5.2 评估：前向–后向算法

评估的目的是求取某个给定序列的出现概率。在需要确定你的模型以多大的概率创建了待建模序列时，评估便十分重要。此外，在确定如序列“主页→主页”的概率是否比序列“主页→注册页”更高时，评估也极为有用。可利用前向 – 后向算法来完成评估。该算法的目标是计算某个隐含状态与观测的概率依赖关系。

> **前向 – 后向算法的数学表示**
>
> 前向 – 后向算法用于在给定某个输出的隐含状态时，计算该输出发生的概率，即 $P(e_k|s)$。乍看上去，这个概率的计算非常困难，因为其中涉及大量概率。如果我们借助链式规则，表达式会迅速扩展。幸运的是，我们可利用一种非常简单的策略来进行求解。
>
> 给定某个观测序列时，e_k 的概率与 e_k 和观测量的联合概率成正比，即
>
> $$p(e_k \mid s) \propto p(e_k, s)$$
>
> 利用概率链式法则，可将 $P(e_k,s)$ 分解为两项的乘积：
>
> $$p(s_{k+1}, s_{k+2}, \cdots, s_n \mid e_k, s_1, s_2, \cdots, s_k)\, p(e_k, s_1, s_2, \cdots, s_k)$$
>
> 这样分解看起来并无明显的益处，然而实际上我们可以将第一个概率表达式中的 $s_1,\cdots,s_k$ 丢弃，因为这类概率是 D **分离的**（D-Separated）。我不打算在这里用太多篇幅讨论 D 分离，但由于我们之前已对模型做过马尔可夫假设，所以可放心地“忘记”这些变量，因为它们都位于我们概率模型中感兴趣的时间点之前，从而有
>
> $$p(e_k \mid s) \propto p(s_{k+1}, s_{k+2}, \cdots, s_n \mid e_k)\, p(e_k, s_1, s_2, \cdots, s_k)$$
>
> 这便是前向 – 后向算法！

我们可将其设想为一条穿过如图 5-4 所示的概率空间的路径。当给定某个特定的输出（如 E2）时，我们可通过查看前向概率和后向概率来计算该输出的概率值。

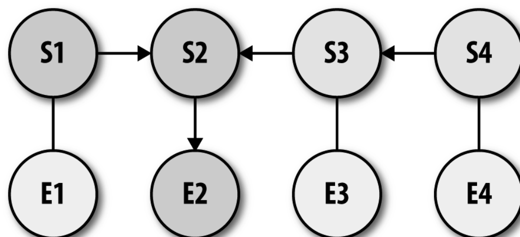


图 5-4：前向 – 后向算法示意

前向项表示的是当给定截至某个时间点 k 的所有输出时，位于点 k 的隐含状态的联合概率。后向项描述的则是给定隐含点时，从 k+1 到最后的输出的条件概率。

利用用户行为

利用之前“主页→注册页→产品页→结账页”的例子，可借助前向－后向算法计算该序列在我们的模型内部发生的概率。首先，构造一个名为 ForwardBackward 的类以对问题进行设定。

```
require 'matrix'
class ForwardBackward
  def initialize
    @observations = ['homepage', 'signup', 'product', 'checkout']

    @states = ['Prospect', 'User', 'Customer']
    @emissions = ['homepage', 'signup', 'product page', 'checkout',
    'contact us']
    @start_probability = [0.8, 0.15, 0.05]

    @transition_probability = Matrix[
      [0.8, 0.15, 0.05],
      [0.05, 0.80, 0.15],
      [0.02, 0.95, 0.03]
    ]

    @emission_probability = Matrix[
      [0.4, 0.3, 0.3], # homepage
      [0.1, 0.8, 0.1], # signup
      [0.1, 0.3, 0.6], # product page
      [0, 0.1, 0.9],   # checkout
      [0.7, 0.1, 0.2]  # contact us
    ]
  end
end
```

这里，我们仅导入了之前所掌握的信息，即转移概率矩阵和输出概率。接下来，我们需要定义前向步，如下所示：

```
class ForwardBackward
  # Initialize
  def forward
    forward = []
    f_previous = {}
    @observations.each_with_index do |obs, i|
      f_curr = {}
      @states.each do |state|
        if i.zero?
          prev_f_sum = @start_probability.fetch(state)
        else
          prev_f_sum = @states.reduce(0.0) do |sum, k|
            sum += f_previous.fetch(k, 0.0) * @transition_probabilit.fetch(k).
```

```
            fetch(state)
          end
        end
        f_curr[state] = @emission_probability.fetch(state).fetch(obs) *
        prev_f_sum
        forward << f_curr
        f_previous = f_curr
      end
    end

    p_fwd = @states.reduce(0.0) do |sum, k|
      sum += f_previous.fetch(k) * @transition_probability.fetch(k).fetch
      (@end_state)
    end

    {
      'probability' => p_fwd,
      'sequence' => forward
    }
  end
end
```

对于每个观测，前向算法会遍历每个状态，并将其相乘以得到一个关于在给定上下文中状态如何转换的前向概率。接下来，我们需要定义后向算法：

```
class FowardBackward
  # initialize
  # forward

  def backward
    backward = []
    b_prev = {}

    %w[None].concat(@observations[1..-1].reverse).each_with_index do
    |x_i_plus, i|
      b_curr = {}
      @states.each do |state|
        if i.zero?
          b_curr[state] = @transition_probability.fetch(state).fetch(@end_state)
        else
          b_curr[state] = @states.reduce(0.0) do |sum, k|
            sum += @transition_probability.fetch(state).fetch(k) *
            @emission_probability.fetch(k).fetch(x_i_plus) * b_prev.fetch(k)
          end
        end
      end

      backward.insert(0, b_curr)
      b_prev = b_curr
    end

    p_bkw = @states.reduce(0.0) do |sum, s|
      sum += @start_probability.fetch(s) * @emission_probability.fetch(s).
      fetch(@observations[0]) * b_prev.fetch(s)
```

```
      end

      {
        'probability' => p_bkw,
        'sequence' => backward
      }
    end
  end
```

后向算法的工作原理与前向算法非常类似，区别仅在于二者行进的方向相反。接着，我们需要同时测试前向和后向算法，以确保二者能够得到相同的结果（否则便说明我们的算法有误）：

```
class FowardBackward
  # initialize
  # forward
  # backward
  def forward_backward
    size = @observations.length
    fwd, p_fwd = forward.values
    bkwd, p_bkw = backward.values

    # 将两部分合并
    posterior = {}
    @states.each do |state|
      posterior[state] = (1..size).map do |i|
        fwd[i][state] * bwkd[i][state] / p_fwd
      end
    end

    return fwd, bkw, posterior
  end
end
```

前向－后向算法之美体现在当它运行时，它实际上就是在对自身进行测试。这听起来非常令人兴奋。该算法能够求解评估问题——但请牢记，这意味着求取某个给定序列的出现概率。接下来，我们将深入探讨解码问题，即求取最优的隐含状态序列。

5.3　利用维特比算法求解解码问题

解码问题最容易描述。给定某个观测序列，我们希望依据已经掌握的证据求取最优的状态路径。这个目标可用数学语言表述为$\pi * = \text{argmax}_{\pi} P(x, \pi)$，其中 π 为状态向量，而 x 为观测值。

为此，我们可利用*维特比算法*（Viterbi 算法）。你可将该算法理解为一种构造最大伸展树的方法。当给定当前状态时，我们试图找出到达下一个状态的最优路径。与其他贪心算法类似，维特比算法也需要遍历所有可能的下一步状态，并从中选择一个最优解。

该过程可表示为图 5-5。

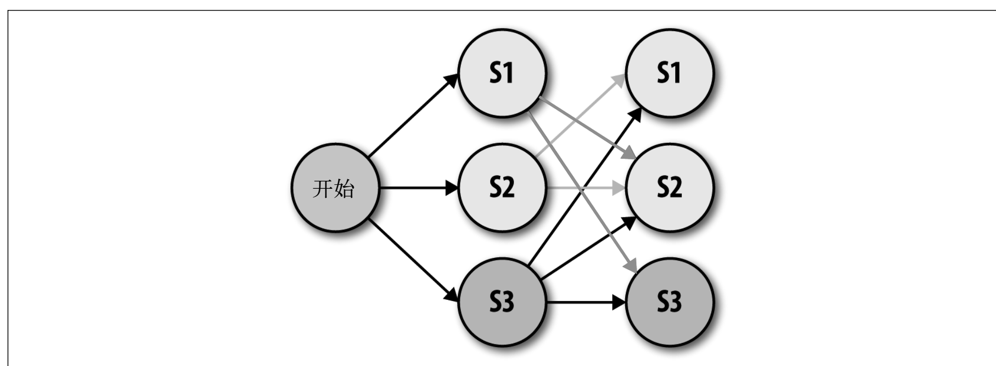


图 5-5：随着时间的推移，一个拥有较高概率的状态将获胜

从上图中可以看出一个状态（如 S1）的相关性如何随时间的推移降低，而状态 S3 的相关性与其他状态相比逐渐增加。图中箭头的明暗表明了概率的减小情况。

我们的目标是通过该算法以最优的方式遍历一个状态集合。为此，我们确定当给定输出，以及从之前的状态转换到当前状态的概率时，某个状态发生的概率。然后，将那这个概率相乘，从而得到序列的出现概率。对整个序列进行迭代，最终便可找到最优序列。

5.4 学习问题

学习问题可能是最容易实现的一个问题。给定状态和观测序列，接下来最有可能发生什么？我们完全可以只依据维特比序列来确定下一步。由于此时尚无输出，我们需要通过最大化下一步的状态概率，以确定接下来的状态。但我们也可求得最可能的输出以及概率最大的状态，它们被称为下一步最优状态输出组合（the next optimal state emission combo）。

如果你对这种求解方式不太了解，请不要担心。在下一节中，我们将更深入地探讨维特比算法的使用。

不幸的是，目前尚无免费和易于获取的数据，来分析给定用户的页面浏览量时用户行为随着时间的变化，但还有另一个类似的问题，我们仅利用隐马尔可夫模型来构建一个词性标注器便可解决它。

5.5 利用布朗语料库进行词性标注

假设给定短语“the quick brown fox”，我们如何对其进行词性标注？我们知道，英语中的词性包括限定词、形容词和名词等。我们可能会将该短语中的各单词分别标注为限定词、形容词、形容词、名词。这个标注例子非常简单，因为我们对于英语语法有基本的了解，

但如何通过算法训练一个模型来完成这样的任务呢？

当然，由于本章是专门介绍隐马尔可夫模型的，所以我们准备利用它来完成最优词性标注。基于对隐马尔可夫模型的了解，对于一个给定的单词序列，我们可利用维特比算法来获得最优的标注序列。本节中，我们将利用布朗语料库，它是第一个电子化的语料库。该语料库中包含了上百万个带有词性标注信息的单词。标注的清单非常长，但请放心，其中仅包含常见的标注，如形容词、名词和动词等。

布朗语料库是利用一种特定类型的标注方法构建的。对于每个单词序列，你会看到一些这样的文字：

```
Most/ql important/jj of/in all/abn ,/, the/at less/ql developed/vbn countries/nns must/md be/be persuaded/vbn to/to take/vb the/at necessary/jj steps/nns to/to allocate/vb and/cc commit/vb their/pp$ own/jj resources/nns ./.
```

在这个例子中，Most 的词性为 ql（表示限定词），important 的词性为 jj（表示形容词），以此类推，直到 `./.`，这表示一个被标注为停止词“.”的英语句号。

这里唯一缺少的是句子的开始字符。一般说来，在我们编写马尔可夫模型时，我们希望得到位置在 t 和 t–1 的单词。由于 most 位于句子最开始，它的前面没有其他单词，因此我们使用一个特殊的名称——START 来表明该序列的开始。这样，我们便可度量从 START 到某个限定词的转移概率。



安装说明

本例所使用的全部代码均可从 Github 获取：https://github.com/thoughtfulml/examples/tree/master/4-hidden-markov-models。

由于 Ruby 处于持续的发展和更新中，因此 README 是了解如何运行这些示例的最佳参考资料。

用 Ruby 来运行本例，不需要其他任何依赖库。

5.5.1 词性标注器的首要问题：`CorpusParser`

词性标注器首先要解决的问题是如何接收数据。最重要的一点是应传递给它恰当的信息，以使词性标注器能够利用数据，并从中学习。在开始之前，我们需要对词性标注器的工作方式做一些假设。我们希望将每次单词标签组合的转移记录在一个二维数组中，并将其包裹在一个名为 `CorpusParser::TagWord` 的简单类中。初始测试代码如下：

```ruby
# test/lib/corpus_parser_spec.rb

require 'spec_helper'

describe CorpusParser do
  let (:stream) { "\tSeveral/ap defendants/nns ./.\n" }
  let (:blank) { "\t    \n" }
  it 'will parse a brown corpus line using the standard / notation' do
    cp = CorpusParser.new

    null = CorpusParser::TagWord.new("START", "START")
    several = CorpusParser::TagWord.new("Several", "ap")
    defendants = CorpusParser::TagWord.new("defendants", "nns")
    period = CorpusParser::TagWord.new(".", ".")

    expectations = [
      [null, several],
      [several, defendants],
      [defendants, period]
    ]

    cp.parse(stream) do |ngram|
      ngram.must_equal expectations.shift
    end

    expectations.length.zero?.must_equal true
  end

  it 'does not allow blank lines from happening' do
    cp = CorpusParser.new

    cp.parse(blank) do |ngram|
      raise "Should never happen"
    end
  end
end
```

这段代码接收了类似于布朗语料库的两个例子，并进行检查，以确保它们可被正确解析。第二个例子是一个全面检查，目的是确保将空行忽略，因为在布朗语料库中空行非常多。

对 CorpusParser 类进行完善，将得到下列代码：

```ruby
# lib/corpus_parser.rb

class CorpusParser
  TagWord = Struct.new(:word, :tag)
  NULL_CHARACTER = "START"
  STOP = " \n"
  SPLITTER = '/'

  def initialize
    @ngram = 2
  end
```

```
  def parse(io)
    ngrams = @ngram.times.map { TagWord.new(NULL_CHARACTER, NULL_CHARACTER) }

    word = ''
    pos = ''
    parse_word = true

    io.each_char do |char|
      if char == "\t" || (word.empty? && STOP.include?(char))
        next
      elsif char == SPLITTER
        parse_word = false
      elsif STOP.include?(char)
        ngrams.shift
        ngrams << TagWord.new(word, pos)

        yield ngrams

        word = ''
        pos = ''
        parse_word = true
      elsif parse_word
        word += char
      else
        pos += char
      end
    end

    unless pos.empty? || word.empty?
      ngrams.shift
      ngrams << TagWord.new(word, pos)
      yield ngrams
    end
  end
end
```

与上一章一样，利用 each_char 来实现解析器是用 Ruby 完成解析任务的最高效方式。现在我们准备进入更有趣的环节：编写词性标注器。

5.5.2 编写词性标注器

词性标注器应当具有如下三种功能：从 CorpusParser 获取数据；存储数据，以便我们计算单词标签组合的概率；计算标签转移的概率。我们希望该类能够告诉我们单词和标签序列的概率值，以及当给定一个纯文本的句子时，确定其最优的标签序列。

为此，我们需要首先解决概率计算问题，其次是给定一个单词序列后，计算其标签序列。最后，我们将实现维特比算法。

我们首先来探讨给定前一位置的单词的标签，如何计算当前单词的词性为某个标签的概

率。利用最大似然估计法，我们可断言该概率值应等于这两个标签同时出现的次数除以前一个标签的出现次数。完成此测试的代码如下：

```ruby
# test/lib/pos_tagger_spec.rb

require 'spec_helper'
require 'stringio'

describe POSTagger do
  let(:stream) { "A/B C/D C/D A/D A/B ./." }

  let(:pos_tagger) {
    pos_tagger = POSTagger.new([StringIO.new(stream)])
    pos_tagger.train!
    pos_tagger
  }

  it 'calculates tag transition probabilities' do
    pos_tagger.tag_probability("Z", "Z").must_equal 0

    # count(previous_tag, current_tag) / count(previous_tag)
    # D与D发生了2次,D发生了3次,因此为2/3
    pos_tagger.tag_probability("D", "D").must_equal Rational(2,3)
    pos_tagger.tag_probability("START", "B").must_equal 1
    pos_tagger.tag_probability("B", "D").must_equal Rational(1,2)
    pos_tagger.tag_probability(".", "D").must_equal 0
  end
end
```

前面我们曾介绍过，序列始于一个隐含标签 START。因此，可以看出，从 D 转移到 D 的概率为 2/3，这是因为从 D 到 D 共出现 2 次，而 D 在该序列中出现了 3 次。为使 `POSTagger` 类能够工作，我们需要编写下列代码：

```ruby
# lib/pos_tagger.rb

class POSTagger
  def initialize(data_io = [])
    @corpus_parser = CorpusParser.new
    @data_io = data_io

    @trained = false
  end

  def train!
    unless @trained
      @tags = Set.new(["START"])
      @tag_combos = Hash.new(0)
      @tag_frequencies = Hash.new(0)
      @word_tag_combos = Hash.new(0)

      @data_io.each do |io|
        io.each_line do |line|
```

```
          @corpus_parser.parse(line) do |ngram|
            write(ngram)
          end
        end
      end
      @trained = true
    end
  end

  def write(ngram)
    if ngram.first.tag == 'START'
      @tag_frequencies['START'] += 1
      @word_tag_combos['START/START'] += 1
    end

    @tags << ngram.last.tag

    @tag_frequencies[ngram.last.tag] += 1
    @word_tag_combos[[ngram.last.word, ngram.last.tag].join("/")] += 1
    @tag_combos[[ngram.first.tag, ngram.last.tag].join("/")] += 1
  end

  # 最大似然估计
  # count(previous_tag, current_tag) / count(previous_tag)
  def tag_probability(previous_tag, current_tag)
    denom = @tag_frequencies[previous_tag]

    if denom.zero?
      0
    else
      @tag_combos["#{previous_tag}/#{current_tag}"] / denom.to_f
    end
  end
end
```

你可能已经注意到，当出现 0 时，我们做了一点错误处理的工作，因为我们将抛出一个 divide-by-zero 的错误。接下来，我们需要解决单词标签组合的概率计算问题，可利用下列已有测试代码来实现：

```
# test/lib/pos_tagger_spec.rb

describe POSTagger do
  let(:stream) { "A/B C/D C/D A/D A/B ./." }
  # 最大似然估计
  # count (word and tag) / count(tag)
  it 'calculates the probability of a word given a tag' do
    pos_tagger.word_tag_probability("Z", "Z").must_equal 0

    # A与B发生了两次,b发生了两次,因此为100%
    pos_tagger.word_tag_probability("A", "B").must_equal 1

    # A与D发生了1次,D发生了3次,因此为1/3
    pos_tagger.word_tag_probability("A", "D").must_equal Rational(1,3)
```

```
    # START与START发生了1次,start发生了1次,因此为1
    pos_tagger.word_tag_probability("START", "START").must_equal 1

    pos_tagger.word_tag_probability(".", ".").must_equal 1
  end
end
```

为使上述测试代码能够在 POSTagger 类中工作，我们还需要编写如下代码：

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability

  # 最大似然估计
  # count (word and tag) / count(tag)
  def word_tag_probability(word, tag)
    denom = @tag_frequencies[tag]

    if denom.zero?
      0
    else
      @word_tag_combos["#{word}/#{tag}"] / denom.to_f
    end
  end
end
```

这样，我们就得到了所期望的 word_tag_probability 和 tag_probability。于是，我们可回答以下问题：给定一个单词和标签序列，该单词取得各标签的概率值是多大？即给定之前单词的标签时当前标签的概率，乘以给定该标签时该单词的概率。完成该功能的测试代码如下：

```
# test/lib/pos_tagger.rb

describe POSTagger do
  it 'calculates probability of sequence of words and tags' do
    words = %w[START A C A A .]
    tags = %w[START B D D B .]
    tagger = pos_tagger

    tag_probabilities = [
      tagger.tag_probability("B", "D"),
      tagger.tag_probability("D", "D"),
      tagger.tag_probability("D", "B"),
      tagger.tag_probability("B", ".")
    ].reduce(&:*)

    word_probabilities = [
```

```
      tagger.word_tag_probability("A", "B"), # 1
      tagger.word_tag_probability("C", "D"),
      tagger.word_tag_probability("A", "D"),
      tagger.word_tag_probability("A", "B"), # 1
    ].reduce(&:*)

    expected = word_probabilities * tag_probabilities

    pos_tagger.probability_of_word_tag(words, tags).must_equal expected
  end
end
```

因此，基本上，我们所计算的是单词标签的概率乘以标签转移的概率。利用下列代码，我们可以轻松地在 PosTagger 类中实现该功能：

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability
  # word_tag_probability

  def probability_of_word_tag(word_sequence, tag_sequence)
    if word_sequence.length != tag_sequence.length
      raise 'The word and tags must be the same length!'
    end
    # word_sequence %w[START I want to race .]
    # Tag sequence %w[START PRO V TO V .]

    length = word_sequence.length

    probability = Rational(1,1)

    (1...length).each do |i|
      probability *= (
        tag_probability(tag_sequence[i - 1], tag_sequence[i]) *
        word_tag_probability(word_sequence[i], tag_sequence[i])
      )
    end

    probability
  end
end
```

至此，我们已经能够确定一个给定单词和标签序列的概率。但如果当给定一个句子和训练数据时，我们还能够确定最优标签序列，就再好不过了。为此，我们需要编写如下简单测试：

```
# test/lib/pos_tagger_spec.rb

describe POSTagger do
```

```
describe 'viterbi' do
  let(:training) { "I/PRO want/V to/TO race/V ./. I/PRO like/V cats/N ./." }
  let(:sentence) { 'I want to race.' }
  let(:pos_tagger) {
    pos_tagger = POSTagger.new([StringIO.new(training)])
    pos_tagger.train!
    pos_tagger
  }

  it 'will calculate the best viterbi sequence for I want to race' do
    pos_tagger.viterbi(sentence).must_equal %w[START PRO V TO V .]
  end
end
end
```

实现该测试的工作量稍多一点，因为维特比算法相对比较复杂。因此我们需要按部就班地进行。第一个问题是，我们的方法需要接收字符串，而非单词序列。我们需要依据空格对字符串进行切割，并将停止字符保留。为此，我们需要编写下列代码来为维特比算法做准备：

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability
  # word_tag_probability

  def viterbi(sentence)
    parts = sentence.gsub(/[\.\?!]/) {|a| " #{a}" }.split(/\s+/)
  end
end
```

维特比算法是一种迭代算法，这意味着它会在每一轮迭代中依据上一轮的答案进行决策。因此，我们需要记忆之前的概率值，并保存最优标签。可利用下列方式初始化并确定最优标签：

```
# lib/pos_tagger.rb

class POSTagger
  # initialize
  # train!
  # write
  # tag_probability
  # word_tag_probability

  def viterbi(sentence)
    # parts

    last_viterbi = {}
```

```
      backpointers = ["START"]

      @tags.each do |tag|
        if tag == 'START'
          next
        else
          probability = (
            tag_probability("START", tag) *
            word_tag_probability(parts.first, tag)
          )

          if probability > 0
            last_viterbi[tag] = probability
          end
        end
      end

      backpointers << (
        last_viterbi.max_by {|k,v| v} ||
        @tag_frequencies.max_by {|k,v| v}
      ).first
    end
  end
```

此时，`last_viterbi` 仅有一个选择，即 `{"PRO"=>1.0}`。这是因为从 START 转移到任何其他标签的概率均为 0。同样，`backpointers` 中将包含 START 和 PRO。因此，既然已经建立好初始化步骤，我们只需对其余部分进行迭代即可：

```
  # lib/pos_tagger.rb

  class POSTagger
    # initialize
    # train!
    # write
    # tag_probability
    # word_tag_probability

    def viterbi(sentence)
      # parts
      # 初始化


      parts[1..-1].each do |part|
        viterbi = {}
        @tags.each do |tag|
          next if tag == 'START'
          break if last_viterbi.empty?

          best_previous = last_viterbi.max_by do |prev_tag, probability|
            (
              probability *
              tag_probability(prev_tag, tag) *
              word_tag_probability(part, tag)
```

```
        )
      end

      best_tag = best_previous.first

      probability = (
        last_viterbi[best_tag] *
        tag_probability(best_tag, tag) *
        word_tag_probability(part, tag)
      )

      if probability > 0
        viterbi[tag] = probability
      end
    end


    last_viterbi = viterbi

    backpointers << (
      last_viterbi.max_by{|k,v| v} ||
      @tag_frequencies.max_by{|k,v| v }
    ).first
  end
  backpointers
 end
end
```

这里我们所做的是仅保存相关信息。如果出现 `last_viterbi` 为空的情况，我们会转而使用 `@tag_frequencies`。仅当修剪过度时才会出现这种情形。但这种方法要比在内存中保存所有信息高效得多。

至此，所有的工作都完成了！但我们应如何对其性能进行评估呢？

5.5.3 通过交叉验证获取模型的置信度

在这个阶段，编写一个交叉验证测试是明智之举。虽然我们使用的是一个非常朴素的模型，但还是希望其准确率至少能够达到 20%。我们将之写入一个 10 份交叉验证规约中。我们并不要求该模型处于一定的置信范围内，而只是将错误率呈现给用户。当我在自己的机器中运行该测试时，得到的错误率约为 30%。我们还将讨论如何改进，但对于我们的目标而言，鉴于仅考察两个概率值，这已经足够好了：

```
# test/cross_validation_spec.rb

require 'spec_helper'

describe "Cross Validation" do
  let(:files) { Dir['./data/brown/c***'] }

  FOLDS = 10
```

```
    FOLDS.times do |i|
      let(:validation_indexes) do
        splits = files.length / FOLDS
        ((i * splits)..((i + 1) * splits)).to_a
      end

      let(:training_indexes) do
        files.length.times.to_a - validation_indexes
      end

      let(:validation_files) do
        files.select.with_index {|f, i| validation_indexes.include?(i) }
      end

      let(:training_files) do
        files.select.with_index {|f, i| training_indexes.include?(i) }
      end

      it "cross validates with a low error for fold #{i}" do
        pos_tagger = POSTagger.from_filepaths(training_files, true)
        misses = 0
        successes = 0

        validation_files.each do |vf|
          File.open(vf, 'rb').each_line do |l|
            if l =~ /\A\s+\z/
              next
            else
              words = []
              parts_of_speech = ['START']
              l.strip.split(/\s+/).each do |ppp|
                z = ppp.split('/')
                words << z.first
                parts_of_speech << z.last
              end

              tag_seq = pos_tagger.viterbi(words.join(' '))
              misses += tag_seq.zip(parts_of_speech).count {|k,v| k != v }
              successes += tag_seq.zip(parts_of_speech).count {|k,v| k == v }
            end
          end
          puts Rational(misses, successes + misses).to_f
        end
        skip("Error rate was #{misses / (successes + misses).to_f}")
      end
    end
  end
```

由此产生的错误率约为 20%~30%。实事求是地说，这不精确。导致这种结果的部分原因在于，布朗语料库对标签的类别进行了细分，因此如果你不在意词性的细微差异（如物主代词和常规代词），错误率会低得多。

5.5.4 模型的改进方案

正如我们所有的程序示例一样，改进该模型的最佳方式是首先确定其性能，并进行迭代。提升该模型性能的一种快捷方式是每次查看多个单词。因此，需要求取给定之前两个标签而非之前的一个标签时，某个标签的概率。可通过修改语料库的标注器来达到这个目的。

但我要说，本例的确已经能够很好地工作，而且非常易于实现！

5.6 小结

在需要依据一些观测数据确定隐含数据时，隐马尔可夫模型是最有趣的候选模型之一。例如，你可确定某位用户的真实状态、找到某个单词的隐含标签，甚至是跟随乐谱。

通过本章的学习，你了解了如何将状态机推广为马尔可夫链，它可用于对系统行为持续进行建模。我们还增加了一个隐藏单元以确定当给定一些容易被观测的输出时，模型的隐含状态。你还了解了使用隐马尔可夫模型的三个阶段分别为评估、解码和学习，以及如何对这些问题进行求解。最后，我们利用布朗语料库和维特比算法搭建了一个词性标注器。

第 6 章

支持向量机

到底是什么决定用户的忠诚度？在商业上，忠诚度通常与经常光顾并消费有关。但如何度量忠诚和不忠诚的决定因素呢？

本章中，我们将利用支持向量机从理论上求解该问题。该算法利用许多特征对象来生成两个类别（如忠诚与不忠诚）之间的决策面。本章的最后，我们将通过一个实际案例介绍如何分析电影评论所体现的情绪。

6.1 求解忠诚度映射问题

在线商店拥有两类顾客：忠诚的顾客和不忠诚的顾客。忠诚的顾客会不断地回头购买某个品牌的商品，而不忠诚的顾客要么只逛不买，要么不关心品牌，消费无度。我们的目标是就订单数量和平均订单金额，确定到底是什么使得顾客忠诚或不忠诚。

设想我们的数据分布如图 6-1 所示。

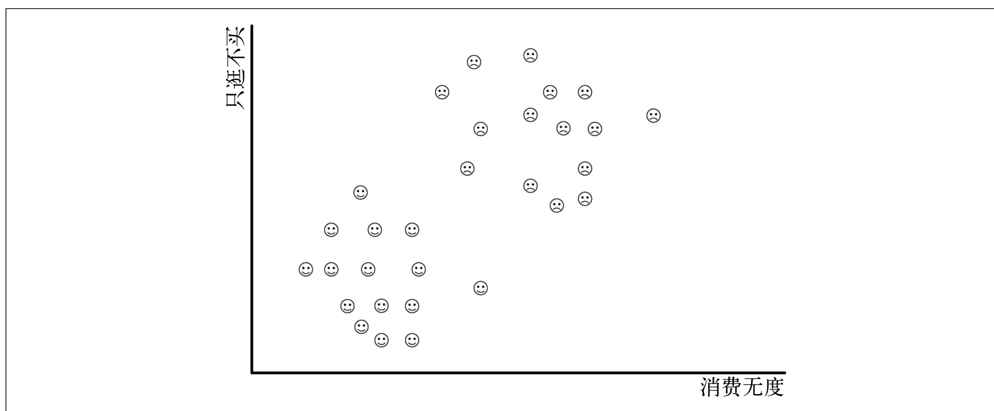


图 6-1：忠诚顾客和不忠诚顾客之间存在明确的界限

由于是首次接触这类问题，我们有多种方式来判定用户是否忠诚。我们可利用 *K* 近邻分类程序（参见第 3 章），这种方法实际上是将数据围绕一个中心形成了一个簇（参见图 6-2）。

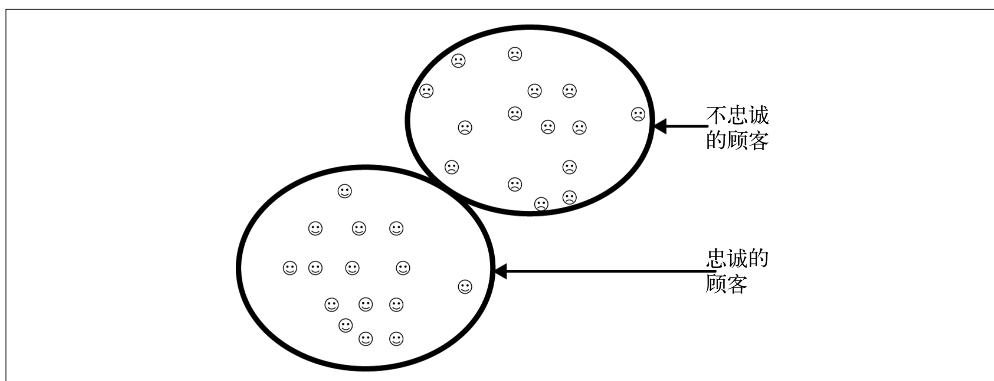


图 6-2：利用聚类，可构建一个如图所示的分类结果

但这并不是我们想要的结果。我们不想了解平均意义上的忠诚用户和不忠诚用户具有什么特点，而是希望找到这两个类别之间的*决策面*。所谓决策面是一条介于两类数据之间的直线。不幸的是，这个过程并不像听起来那样简单，因为我们可以画出无数条决策面。

幸运的是，有一个算法可帮助我们求解该问题——支持向量机（Support Vector Machine，SVM）。SVM 最早由 Vladimir Vapnik 于 20 世纪 80 年代提出，被作为一种数据分类方法。该方法的现代解释（http://link.springer.com/article/10.1007%2FBF00994018）在 1995 年被提出，在严格性上与提出时相比有所放松。最初提出 SVM 是为了帮助求解两类分类问题。这些类型的问题可以是布尔型（真、假）、ids（3,4）或正负（1,–1）。SVM 的特别之处在于，它在高维空间中也有很好的表现，可避免维数灾难。此外，其计算效率也较高。

该算法并不是从两个数据集合之间任意选择一条直线，而是要使它们之间的距离最大化（参见图 6-3）。例如，在我们的忠诚顾客与不忠诚顾客问题中，我们可得到这两个类别之间的最优决策面。确定了决策面的位置之后，便可回答是什么使得顾客忠诚或不忠诚。

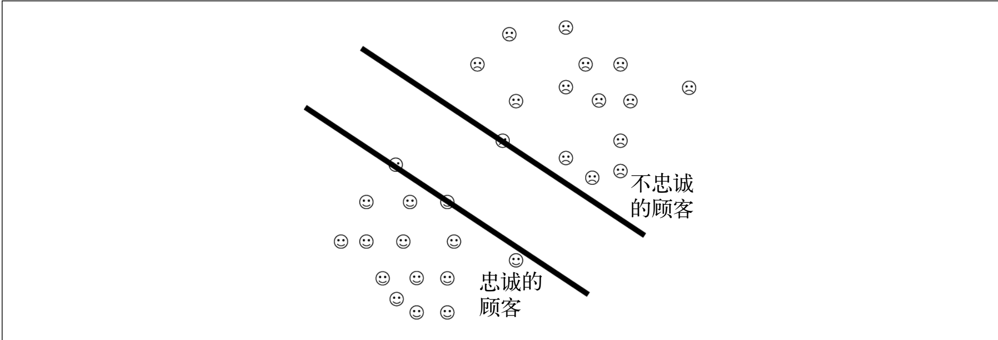


图 6-3：利用 SVM，可利用间隔将两组数据分离

6.2 SVM的推导过程

从概念上看，我们理解 SVM 使两个集合之间的距离最大化，但它是如何做到这一点的呢？

我们的目标是求解方程 $wx-b=0$（参见图 6-4）中的 w。该方程可重新表示为$b=\sum_{i=1}^{n} w_i x_i$，其中 x_i 为数据的第 i 维特征，而 w_i 待定。这些方程在 n 维空间中定义了一个超平面。*超平面*（hyperplane）即 n 维直线。最重要的是应意识到，我们是在两个集合之间确定一个超平面，并确定最远的数据点之间的距离。

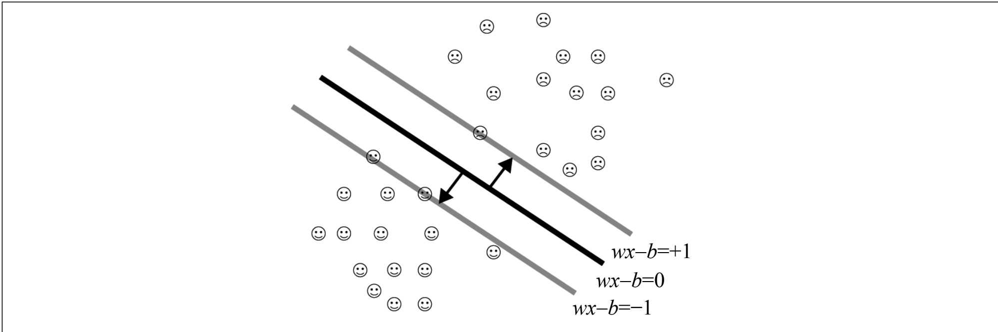


图 6-4：$wx-b=0$ 定义了将两类数据分离的超平面

但这只是两类数据之间的一个任意空间。我们希望找到能够将两个集合最大化分离的空间。为此，我们需要定义两个分别位于已知超平面上方和下方的超平面。在本例中，我们可将位于上方的超平面定义为 $wx+b=1$，而将下方的超平面定义为 $wx+b=-1$。此时，我们

对实际数据尚一无所知，但在已知直线上方和下方各有一个间隔。

既然定义了三个超平面，我们希望将上方超平面和下方超平面的间隔最大化。我们采用几何方法进行推导。现有两条平行线，我们需要求出二者的距离。唯一的方法是找到一个沿着与所有超平面垂直的方向移动的超平面。在本例中，上方超平面与下方超平面之间的垂直线段的长度恰为$\frac{2}{\|w\|}$。我们的初始目标是将间隔（两超平面之间的欧氏距离）最大化，但我们可将其简化为最小化$\|w\|$。

该方法具有许多优点，其中之一是$\|w\| = \sqrt{ww}$（意味着这是一个凸函数）。借助许多已有算法，可对凸函数快速求解。然而，我们还有一个更好的选择——将目标函数重新定义为$\frac{1}{2}\|w\|^2$，这个新的优化问题被称为*二次规划*（quadratic program）。利用 Karush-Kuhn-Tucker 条件，我们可轻松地求解这个问题。

现在我们已经掌握了一种将两个数据集的间隔最大化的方法。现在问题成为了一个简单的二次规划问题，而且可快速求解。但还有一个问题有待解决——当数据不是线性可分时。

6.3 非线性数据

你可能已经注意到，在之前的所有例子中，数据都是线性可分的。然而，大多数数据都不是线性的，而且维数较高。我们希望能够将其变换到更加稳健和非线性的空间中。可利用核技巧借助 SVM 来实现这种变换。这实际上是将线性模型转换为非线性，并解决许多与非线性数据相关的问题。

6.3.1 核技巧

假设给定某一线性不可分数据集（如图 6-5 所示）。

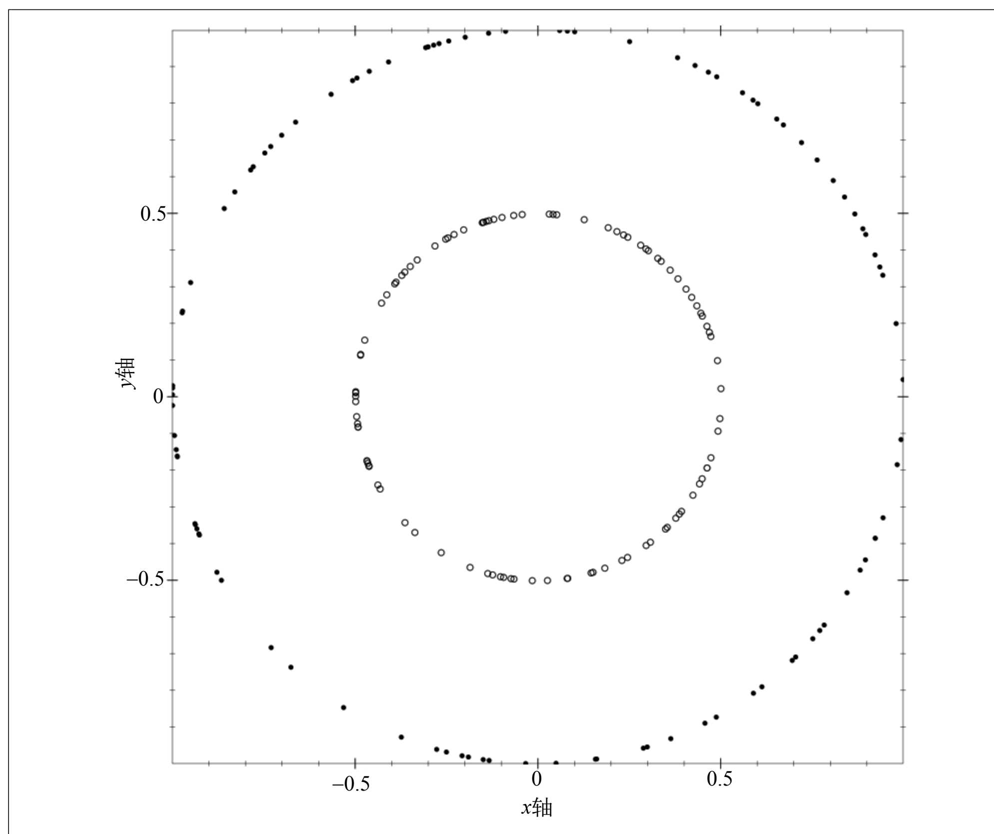


图 6-5：位于一个圆内的圆无法通过直线进行分离

你注意到我们无法通过一条直线将这两组数据分离。为将二者分离，我们只能借助非线性决策面，如圆形。利用任意类型的线性决策面，我们都无法将这两个类别的数据划分为两个圆形。幸运的是，我们可使用一种技巧来克服这个难点！*核技巧*只是将数据由一种类型变换为另一种类型。例如，我们不在二维空间中观察这个圆形，而是将 $<x, y>$ 变换到 $\langle x^2, \sqrt{2}\,xy, y^2 \rangle$，情况会怎样？变换后的数据集如图 6-6 所示。

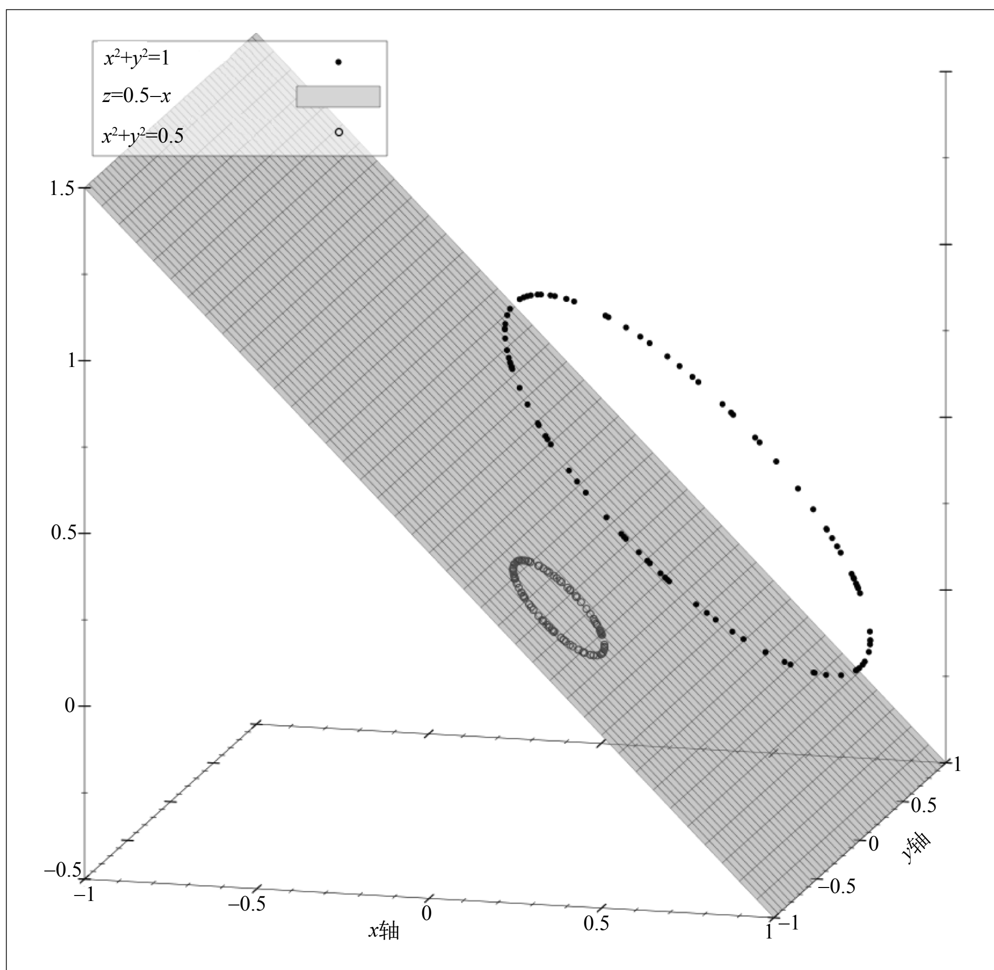


图 6-6：用不同的方式查看同一组数据

这便是核技巧的梗概。大体上说，它接收一种维度的数据，然后将其变换到更高维的空间中，这样我们便可利用直线将它们分离。在上面这个例子中，我们使用的核函数称为多项式核。

从数学上看，核技巧接收一个线性的 x，然后将其转换为 $\phi(x)$ 或某个更适合该数据集的 x 的函数。然而，在利用核技巧时，需要考虑以下几点。

- $\phi(x_i)\dot{\phi}(x_j)$ 会被多次计算，这可能会花费大量时间。
- ϕ 本身可能就是一个难于计算的复杂函数。
- 如果训练数据的规模很大，则通常利用核技巧会大大增加计算开销。

因此，我们并不是将所有的 x 都变换为 $\phi(x)$，而是尝试一些不同的计算，如 $k(x_i, x_j) = \phi(x_i) * \phi(x_j)$。我们并不只是将原数据点 x 变换为新函数，我们映射的是 $\phi(x)$ 的内积。

我们还需要一个易于计算的量。好消息是这样的量的确存在！它被称为核函数。这种类型的函数已经隐含计算了内积，因此它避免了计算内积这个步骤，因此对计算效率而言是一种优化。

下列核函数都拥有 $K(x, y) = \langle\phi(x), \phi(y)\rangle$ 这样的优良性质：

- 齐次多项式
- 非齐次多项式
- 径向基函数

1. 齐次多项式

最简单的核函数是齐次多项式：

$$K(x_i, x_j) = (x_i^{\mathrm{T}} x_j)^d$$

其中，d 为多项式的阶数，它可取任意大于 0 的整数。由于该函数计算开销低，且易于计算，故在手写体检测中得到了广泛应用。

多项式核之所以有用，是因为它利用了相似性以及数据的组合。以二阶多项式核为例：

$$K(x, y) = \left(\sum_{i=1}^{n} x_i y_i\right)^2 = \sum_{i=1}^{n} x_i^2 y_i^2 + \sum_{i=2}^{n} \sum_{j=1}^{i-1} \sqrt{2}\, x_i y_i \sqrt{2}\, x_j x_i$$

虽然这看起来有些复杂，但其实蕴含了一些很有趣的内容。其中不但有我们可以预期的交互项 $x_i^2 y_i^2$，而且还有组合项 $x_i y_i * x_j y_j$。当某些维度应当被划分为一组时，这是极为有用的。

请注意图 6-6 展示的是一个齐次多项式，它的阶数为 2，这个阶数在大多数情况下都具有优异的表现。

2. 非齐次多项式

非齐次多项式与齐次多项式非常类似，区别仅在于它引入了一个非负常量 c（严格大于 0）。

非齐次多项式的一般形式如下：

$$K(x_i, x_j) = (x_i^{\mathrm{T}} x_j + c)^d$$

这里增加的常量 c 提升了高阶而非低阶特征的相关性。

3. 径向基函数

较之上述两个多项式核函数，径向基函数（Radial Basis Function，RBF）的使用频率往往更高，这要归功于它在高维空间中的优秀表现。径向基函数的一般形式为：

$$K(x_i, x_j) = \exp\left\{-\frac{\| x_i - x_j \|_2^2}{2\sigma^2}\right\}$$

该式的分子为两点欧氏距离的平方，而 σ 是一个自由参数。

不幸的是，我们缺乏将径向基函数可视化的有效途径。这里要注意的是，径向基核函数实际上创建了一个无穷维空间，而齐次多项式仅将原始空间的维数增加了一维。这实际上是 RBF 的一个突出特性，因为在使用它时，你已经克服了与维数灾难有关的问题。

4. 核函数的选择

选择使用非齐次多项式还是 RBF，这是很复杂的。很多时候，SVM 库都会将 RBF 作为默认的核函数。但这并不能作为使用某种核函数的理由。一篇文章（http://cbio.ensmp.fr/~jvert/svn/bibli/local/Smola1998connection.pdf）指出，这些核函数可对数据归一化，大体上相当于施加了一个低通滤波器。

从概念上看，二阶多项式核会将二维数据点变换至三维空间，并试图拟合出一个最平坦的面。虽然 RBF 和多项式核时常看起来很有必要使用，然而它们实际上比较容易对数据产生过拟合，因此请谨慎使用。

6.3.2 软间隔

到目前为止，我们所讨论的内容都面临着一个挑战，即数据集需要完全线性可分。设想这样一种情况：我们的客户只是表面上看起来忠诚，但其实并非真的忠诚（如图 6-7 所示）。

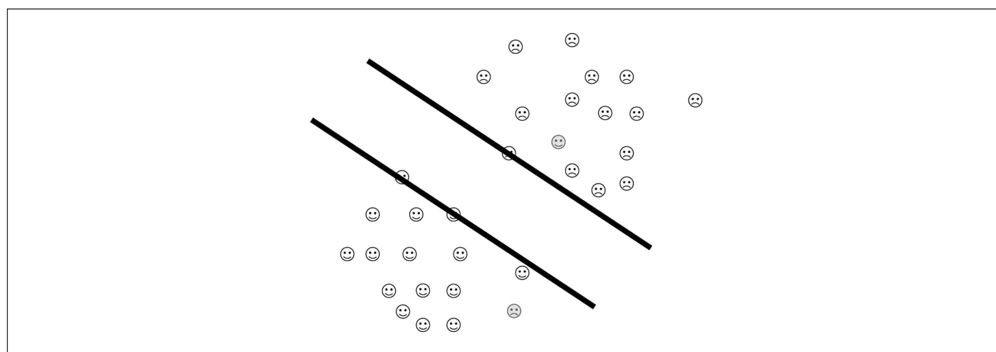


图 6-7：出现误差时的情形，说明了松弛变量的重要性

在该图中，我们的不忠诚客户位于忠诚阵营。该数据集仍然是线性可分的，但存在少量分

类错误。我们并不试图寻找完美的核函数，而是应将这些错误忽略。在 Vladimir Vapnik 当年提出 SVM 时，虽然所做的假设是数据要么直接线性可分，要么用某个核函数变换后线性可分。但在大多数情况下，数据都没有这样完美，因此需要采取一些手段来进行优化：

- 用松弛变量优化
- 引入一个新参数 C

1. 用松弛变量优化

在我们之前的例子中，数据都是线性可分的，但如果数据集不具有线性可分性，则之前的优化策略将失效。好在，对于这样的数据，有一种优雅的优化策略——引入一个松弛变量（slack variable）。

我们不再单纯地优化 $\frac{1}{2}\| w \|^2$，而是引入一组变量 ζ_i 来对误差进行权衡。这些松弛变量实际上对应间隔误差（marginal error）。我们也希望将间隔误差最小化，因此可将其作为惩罚项添加到之前的优化问题中，即 $\frac{1}{2}\| w \|^2 + \sum_{i=1}^{n} \zeta_i$。实际上，我们是将误差参数添加到之前的最小化问题中。

2. 利用C权衡松弛变量最小化与间隔最大化

在之前的最小化问题中，你可能注意到，我们赋予间隔最大化（即 $\frac{1}{2}\| w \|^2$）的权重与松弛变量之和（或间隔误差）的权重是等同的。我们知道，在机器学习问题中处处需要作出权衡，因此该方法是否有效存在一些不确定性。Vapnik 所采取的策略是引入一个对松弛变量进行加权的复杂性参数。该参数由用户定义，对松弛变量而非原始的间隔最大化问题进行加权。

现在，我们的最小化函数就变为：

$$\frac{1}{2}\| w \|^2 + C\sum_{i=1}^{n} \xi_i$$

实践中，这个复杂性参数需要取为正实数，通常可借助交叉验证来确定。在下一节中，我们将通过一个利用 SVM 进行情绪分析的案例，深入探讨如何找到合适的复杂性参数。

6.4 利用SVM进行情绪分析

假设你需要对一系列单词进行情绪分析。如何实现？语言中有很多语境线索、幽默讽刺以及复杂性。例如像“Let’s get stupid”这样的句子，其情绪就颇为模糊，因为它既可表达正面情绪，也可表达负面情绪。

在本节中，我们将搭建一个用于确定电影评论中的情绪的分析系统。下面首先以类图的形式讨论这个工所应具备的特点。在确定该工具的构成之后，我们将构建一个 `Corpus` 类、一个 `CorpusSet` 类以及 `SentimentClassifier` 类。其中，`Corpus` 类和 `CorpusSet` 类将文本转化为数值型信息。在 `SentimentClassifier` 中，我们将利用 SVM 算法构建情绪分析器。



安装说明

本例中使用的所有代码均可从 GitHub 获取（https://github.com/thoughtfulml/examples/tree/master/5-support-vector-machines）。

由于 Ruby 处于持续的发展和更新中，因此 README 是了解如何运行这些示例的最佳参考资料。

用 Ruby 来运行本例，不需要其他任何依赖库。

6.4.1 类图

我们的工具将接受一组包含正面或负面情绪的训练单词和句子。利用这些句子，我们可构建一个信息语料库。在 `Corpus` 类中，我们有偏向正面情绪的文本，或偏向负面情绪的文本，这些文本用停顿符号分隔，如"!""?"或"."。一旦构建起负面情绪或正面情绪文本语料库，我们便需要将其整合到一个 `CorpusSet` 类中。基本做法是将两个或更多 `CorpusSet` 对象存放到一个对象中。

从此，`CorpusSet` 类将由 `SentimentClassifier` 类所使用，以在将来进行情感分析。图 6-8 展示了本案例所对应的类图。

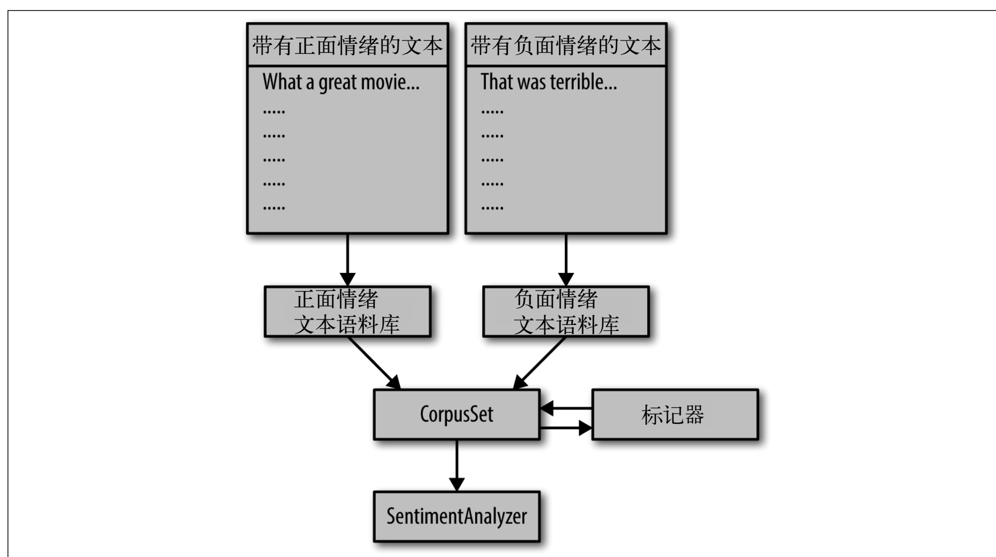


图 6-8：关于如何用 SVM 分析情感的一般性类图

> **corpus 和 corpora 的含义**
>
> corpus 与 corpse 类似，都表示由大量东西构成的集合，但在本例中，它表示由大量文字构成的集合。这个词在自然语言处理领域被广泛使用，表示人们以前撰写的、可用于知识推理的大量文字。在本例中，我们用 corpus 表示由带有某种情绪的大量文本所构成的集合。
>
> corpora 仅为 corpus 的复数形式。

6.4.2 Corpus类

我们的 `Corpus` 类将包含下列功能：

- 符号化文本
- 情绪倾向，即某段文字是 `:negative`，还是 `:positive`
- 将情绪倾向映射为一个实值
- 从语料库中返回一个无重复元素的单词集

1. 文本的符号化

通过第 4 章我们了解到，符号化文本有多种方式，如词干、字母频率、表情符号和单词的提取（参见图 6-9）。在这里，我们仅将单词符号化。它们被定义为非字母字符之间的字符串。因此，从像“The quick brown fox.”这样的句子中，我们可提取出 the、quick、brown、fox。我们并不关心标点符号，并且希望将 Unicode 编码的空格和非单词字符忽略。

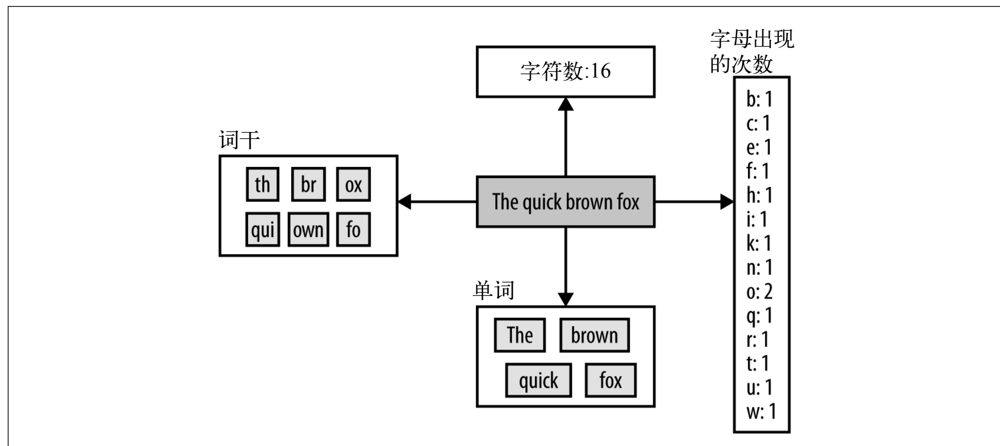


图 6-9：有多种方式可将句子和文本符号化

符号化功能的测试程序如下所示：

```ruby
# test/lib/corpus_spec.rb

require 'spec_helper'
require 'stringio'

describe Corpus do
  describe "tokenize" do
    it "downcases all the word tokens" do
      Corpus.tokenize("Quick Brown Fox").must_equal %w[quick brown fox]
    end

    it "ignores all stop symbols" do
      Corpus.tokenize("\"'hello!?!?!.'\"  ").must_equal %w[hello]
    end

    it "ignores the unicode space" do
      Corpus.tokenize("hello\u00A0bob").must_equal %w[hello bob]
    end
  end
end
```

至此，我们便可构建类方法 ::tokenize 来实际处理我们的测试用例：

```ruby
# lib/corpus.rb

class Corpus
  def self.tokenize(string)
    string.downcase.gsub(/['"\.\?\!]/, '').split(/[[:space:]]/)
  end
end
```



我们使用了 [[:space]]，它在 Ruby 中表示我们将按照 Unicode 和非 Unicode 编码的空格进行单词切割。

虽然你可以投入大量时间来优化这个符号化步骤，但对本例而言，效果已经足够好了。在机器学习中，获取更多的数据通常要胜过对若干小的细节进行优化。

2. 情绪倾向：:positive还是:negative

对于每个 Corpus，我们需要为之依附一个情绪倾向，即偏向正面还是偏向负面。在大多数情形中，我们可选择一个任意实数，如 –1 或 +1。但鉴于其任意性，我们应当使用一些特定的符号。在这里，我们拟采用符号 :positive 和 :negative。

由于我们只关心符号与语料库的情绪标注信息是否一致，因此可进行下列测试：

```ruby
# test/lib/corpus_spec.rb

describe Corpus do
```

```
  let(:positive) { StringIO.new('loved movie!! loved') }
  let(:positive_corpus) { Corpus.new(positive, :positive) }

  it 'consumes a positive training set' do
    positive_corpus.sentiment.must_equal :positive
  end
end
```

为使该类正常工作，我们需要构建一个可接收 `file` 和 `sentiment` 并为 `:sentiment` 生成 `attr_reader` 的骨架类：

```
# lib/corpus.rb

class Corpus
  # 符号化

  attr_reader :sentiment
  def initialize(file, sentiment)
    @file = file
    @sentiment = sentiment
  end
end
```

这是一种非常简单的方法。请注意，我们还为 `Corpus` 类引入了一个 `file` 参数，它指向我们的训练文件。

3. 情绪代码

你可能已经看出，目前的这些代码存在一个问题。你可将任何内容作为情绪传入，且它没有被映射为任何数值型情绪值（无论是 –1 还是 1）。如果我们希望使用 SVM 算法，则这样是没有任何用处的。我们应当对正确和错误两种情况进行测试：

```
# test/lib/corpus_spec.rb

describe Corpus do
  it 'defines a sentiment_code of 1 for positive' do
    Corpus.new(StringIO.new(''), :positive).sentiment_code.must_equal 1
  end

  it 'defines a sentiment_code of 1 for positive' do
    Corpus.new(StringIO.new(''), :negative).sentiment_code.must_equal -1
  end
end
```

在训练 SVM 时，我们将使用这种简化版的映射方法。为此，我们可编写下列代码：

```
# lib/corpus.rb

class Corpus
  # 初始化
  # 符号化
```

```
  attr_reader :sentiment

  def sentiment_code
    {
      :positive => 1,
      :negative => -1
    }.fetch(@sentiment)
  end
end
```

6.4.3 从语料库返回一个无重复元素的单词集

现在只剩最后一步，即从训练文件中返回一个无重复元素的单词集合。这一步的目标是将位于语料库中的单词返回，以便它们能够被压缩到 CorpusSet 类中。由于在大多数情况下，我们都假定文件是按字符串读入的，因此我们可将 StringIO 对象视为文件，由此减少了写入临时文件的需求：

```
# test/lib/corpus_spec.rb

describe Corpus do
  let(:positive) { StringIO.new('loved movie!! loved') }
  let(:positive_corpus) { Corpus.new(positive, :positive) }

  it 'consumes a positive training set and unique set of words' do
    positive_corpus.words.must_equal Set.new(%w[loved movie])
  end
end
```

至此，我们需要在 Corpus 上实现 #words 方法。该方法应返回被传入的单词集合。其具体实现如下：

```
# lib/corpus.rb

class Corpus
  # 初始化
  # 符号化
  # sentiment_code

  attr_reader :sentiment

  def words
    @words ||= begin
      set = Set.new
      @io.each_line do |line|
        Corpus.tokenize(line).each do |word|
          set << word
        end
      end
      @io.rewind
      set
```

```
    end
  end
end
```

既然我们已经实现了符号化，并将所有单词存入一个无重复元素的集合，现在可只关注我们的工具的接口：CorpusSet 类。

6.4.4 CorpusSet类

现在，我们需要定义 CorpusSet 类。该类可接收多个 Corpus 对象，将它们压缩到一个单词集合中，并为将要使用的 SentimentClassifier 对象构建一个向量。这正是数据和支持向量机之间的接口部分。我们再次强调，CorpusSet 类将负责以下任务：

(1) 将两个 Corpus 对象整合到一起；
(2) 构建一个关于两个语料库的稀疏向量。

1. 整合两个语料库对象

我们的第一个测试将接受两个 Corpus 对象，并将它们整合到一个 CorpusSet 类中。具体代码如下所示：

```
# test/lib/corpus_set_spec.rb

require 'spec_helper'

describe CorpusSet do
  let(:positive) { StringIO.new('I love this country') }
  let(:negative) { StringIO.new('I hate this man') }

  let(:positive_corp) { Corpus.new(positive, :positive) }
  let(:negative_corp) { Corpus.new(negative, :negative) }

  let(:corpus_set) { CorpusSet.new([positive_corp, negative_corp]) }

  it 'composes two corpuses together' do
    corpus_set.words.must_equal %w[love country hate man]
  end
end
```

该测试接受两个不同的 Corpus 对象，并将其混合到一个单词集中。CorpusSet 类的具体实现如下：

```
# lib/corpus_set.rb

class CorpusSet
  attr_reader :words

  def initialize(corpora)
    @corpora = corpora
```

```
    @words = corpora.reduce(Set.new) do |set, corpus|
      set.merge(corpus.words)
    end.to_a
  end
end
```

这是一种非常简单的思路，即将多个集合合并到一个较大的集合中。但现在我们需要定义与 `SentimentClassifier` 相关的接口。

2. 构建与`SentimentClassifier`相关的稀疏向量

至此，我们拥有了一个位于 `CorpusSet` 内的单词集，但我们还需要将它们翻译为支持向量机可使用的数据。最常见的方法是将输入字符串转换为一个由 0 和 1 构成的向量。例如，假设我们有一个语料库“The quick brown fox”，我们希望确定“the fox”对应的向量。第一步应当是读取“the quick brown fox”，然后将其拆分成表 6-1 所示的索引。

表6-1：`CorpusSet`单词

单词	索引
the	0
quick	1
brown	2
fox	3

既然我们已经了解到每个单词所关联的索引，我们可取字符串“the fox”，并生成一个新的行向量，如表 6-2 所示。

表6-2：行向量

单词	the	quick	brown	fox
索引	0	1	2	3
“the fox”	1	0	0	1

因此，对于“the fox”，我们仅将索引 0 和 3 设为 1。这虽然看起来是一种不错的想法，但你应意识到大多数时候，一个由训练数据构成的语料库中可能会包含上千个单词和索引。因此对于每个字符串，我们的行向量中的 0 元素将超过 90%，而 1 不到 10%。故我们应当考虑在 Ruby 中使用稀疏向量或散列。

稀疏向量与矩阵

稀疏向量是一种用于存储向量或矩阵的压缩技术。例如，我们有用 Ruby 描述的下列向量：

```
require 'objspace'
sparse_array = 30_000.times.map {|i| (i % 3000 == 0) ? 1 : 0}
sparse_array.size #=> 30000
ObjectSpace.memsize_of(sparse_array) #=> 302,672 bytes
```

该数组的长度为 30 000，其中有 29 990 个元素都为 0。我们并不存储所有的 0 元素，而是可将该数组变换为一个散列，仅存储与非零元素对应的索引值：

```
sparse_hash = Hash.new(0)

sparse_array.each_with_index do |val, i|
  if val.nonzero?
    sparse_hash[i] = val
  else
    # Skip
  end
end

sparse_hash.size #=> 10
ObjectSpace.memsize_of(sparse_hash) #=> 616 bytes
```

请注意数量显著减少了。大小从最初的 30 000 降到了 10！稀疏向量也可推广到矩阵形式：

```
require 'matrix'
require 'objspace'
matrix = Matrix.build(300, 100) do |row, col|
  if  row % 3 == 0 && col % 300 == 0
    1
  else
    0
  end
end

matrix.row_size * matrix.column_size #=> 30000

# 仅当它是一个类似于数组的对象时,memsiz_of方工作

sparse_matrix = Hash.new(0)

matrix.each_with_index do |e, row, col|
  if e.nonzero?
    sparse_matrix[[row, col]] = e
  else
    # e为0,因此跳过
  end
end

sparse_matrix.length #=> 100
ObjectSpace.memsize_of(sparse_matrix) #=> 5,144 bytes
```

在需要考虑内存占用和计算开销的场合，使用稀疏向量是非常必要的。此时不需要存储占据的空间超过所需的东西。

下面我们用稀疏散列而非向量，构建一个接缝测试来确保我们的情绪分析器从 CorpusSet 中接收到了恰当的信息。测试代码如下：

```ruby
# test/lib/corpus_set_spec.rb

describe CorpusSet do
  it 'returns a set of sparse vectors to train on' do
    expected_ys = [1, -1]
    expected_xes = [[0,1], [2,3]]
    expected_xes.map! do |x|
      Libsvm::Node.features(Hash[x.map {|i| [i, 1]}])
    end

    ys, xes = corpus_set.to_sparse_vectors

    ys.must_equal expected_ys

    xes.flatten.zip(expected_xes.flatten).each do |x, xp|
      x.value.must_equal xp.value
      x.index.must_equal xp.index
    end
  end
end
```

其具体实现如下：

```ruby
# lib/corpus_set.rb

class CorpusSet
  # 初始化

  attr_reader :words

  def to_sparse_vectors
    calculate_sparse_vectors!
    [@yes, @xes]
  end

  private
  def calculate_sparse_vectors!
    return if @state == :calculated
    @yes = []
    @xes = []
    @corpora.each do |corpus|
      vectors = load_corpus(corpus)
      @xes.concat(vectors)
      @yes.concat([corpus.sentiment_code] * vectors.length)
    end
    @state = :calculated
  end

  def load_corpus(corpus)
    vectors = []
    corpus.sentences do |sentence|
      vectors << sparse_vector(sentence)
    end
    vectors
  end
end
```

现在，CorpusSet 便可接收多个 Corpus 对象，并将其转换为信息的稀疏散列，供 SentimentClassifier 使用。这是实际使用 SVM 算法并依据数据进行训练的起点。

6.4.5 SentimentClassifier类

既然我们的应用已经拥有了从正面和负面情绪文本中抽取训练数据的功能，接下来便可构建应用的 SVM 部分（SentimentClassifier 类）。该类的关键在于从一个 CorpusSet 对象接收信息，并将其转换到 SVM 模型中。之后，我们便可将新的信息映射为 :positive 或 :negative。除了构建 SentimentClassifier 类，我们还需解决下列问题。

- 我们需要一些手段重构与 CorpusSet 的交互，因为 CorpusSet 的 API 过于复杂，不易使用。
- 我们需要一个库来实现 SVM 算法。
- 我们需要一些数据来训练和交叉验证。

1. 重构与CorpusSet的交互

SentimentClassifier 类接收一个参数，即 CorpusSet。它表示所有训练数据构成的语料集。不幸的是，由于我们的参数是 CorpusSet 类型，我们可能会使用下列语法：

```
# lib/sentiment_classifier.rb

class SentimentClassifier
  def initialize(corpus_set)
    # 初始化
  end
end

positive = Corpus.new(positive_file_path, :positive)
negative = Corpus.new(negative_file_path, :negative)
corpus_set = CorpusSet.new([positive, negative])
classifier = SentimentClassifier.new(corpus_set)
```

这并不是一种良好的 API 设计——为构建一个 Corpus 对象和一个 CorpusSet 对象，它需要大量之前的信息。实际中，我们更希望利用像工厂方法那样的东西来构建 SentimentClassifier 对象。.build 方法可接收多个指向训练数据的参数。我们并不传入一个散列，而是假设正面情绪文本的扩展名为 .pos，而负面情绪文本的扩展名为 .neg。

生成一个名为 .build 的工厂方法会十分有帮助，且不需要我们显式构建任何对象；它依赖于文件系统类型，因此我们只需对空白处进行填充即可：

```
# lib/sentiment_classifier.rb

class SentimentClassifier
  def self.build(files)
    mapping = {
      '.pos' => :positive,
      '.neg' => :negative
```

```
    }

    corpora = files.map { |file| Corpus.new(file, mapping.fetch(File.extname(fil
e)) }
    corpus_set = CorpusSet.new(corpora)

    new(corpus_set)
  end
end
```

在这个阶段，我们还需要做出两个决策：用什么库来构建 SVM 模型，以及从哪里获取训练数据。

2. 实现支持向量机算法的库：LibSVM

当提及支持向量机的库时，大多数人都会选择 LibSVM。在同类库中，它历史最悠久，完全用 C 语言实现，且拥有很多绑定版本，包括 Python、Java 和 Ruby 等。这里需要提醒你的是，服务于 LibSVM 的 Ruby gem 包很少，而且并非所有 gem 包都有优异的性能。我们将要采用的 rb-libsvm 包支持稀疏向量，因此非常适于用来解决我们的问题。虽然也有一些 gem 包可利用 swig 适配器，但不幸的是，它们也不支持稀疏矩阵。

3. 训练数据

到目前为止，我们尚未讨论我们的工具所使用的训练数据。我么你需要一些可被映射为正面情绪或负面情绪的文本。这些数据需要被组织成若干行，并保存在一些文件中。目前，有很多不同的数据源，但我们所要使用的数据来自 GitHub（https://github.com/jperla/sentiment-data）。这是一个由 Pang Lee 所整理的与影评所体现的情绪有关的数据集。

这个数据集具有高度的特定性，只适合于来自 IMDb（Internet Movie Database）的影评，但对于我们的目的而言，已经足够了。如果你准备在其他程序中使用这些数据，建议你选择与所要解决的问题匹配的数据集。例如，Twitter 的推文的情绪分析应当来自实际的、可被映射为正面情绪和负面情绪的推文。请牢记，通过创建调查问卷和将工作分配给亚马逊旗下的机械土耳其机器人（Mechanical Turk）服务，构建自己的数据集并非想象中那么难。

4. 用影评数据进行交叉验证

交叉验证是确保数据被妥善用于训练以及模型如期工作的最佳途径。其基本思想是将一个较大的数据集划分为两个或更多的子集，然后用其中一个子集做训练，而用其他子集来做验证。

用测试形式，交叉验证如下所示：

```
# test/cross_validation_spec.rb

describe 'Cross Validation' do
  include TestMacros

  def self.test_order
```

```
    :alpha
  end

  (-15..15).each do |exponent|
    it "runs cross validation for C=#{2**exponent}" do
      neg = split_file("./config/rt-polaritydata/rt-polarity.neg")
      pos = split_file("./config/rt-polaritydata/rt-polarity.pos")

      classifier = SentimentClassifier.build([
        neg.fetch(:training),
        pos.fetch(:training)
      ])

      # 找到最小值

      c = 2 ** exponent
      classifier.c = c

      n_er = validate(classifier, neg.fetch(:validation), :negative)
      p_er = validate(classifier, pos.fetch(:validation), :positive)
      total = Rational(n_er.numerator + p_er.numerator, n_er.denominator + p_er.d
enominator)

      skip("Total error rate for C=#{2 ** exponent} is: #{total.to_f}")
    end
  end
end
```

这里我们使用了 skip 和 self.test_order。skip 方法用于向我们提供信息，但本身并不进行任何测试。因为我们试图寻找最优 C，所以只使用测试进行实验。还应注意的是，我们覆盖了 test_order，并将其设为 alpha。这是因为最小测试默认使用随机顺序，这意味着当我们遍历从 –15 到 15 这个数列时，将得到无序的数据。如果我们按顺序观察数据，则结果将更加易于解释。

另外需要注意的是，我们引入了两个新方法：split_file 和 validate。它们都位于我们的测试宏模块中：

```
# test/test_macros.rb

module TestMacros
  def validate(classifier, file, sentiment)
    total = 0
    misses = 0

    File.open(file, 'rb').each_line do |line|
      if classifier.classify(line) != sentiment
        misses += 1
      else
      end
      total += 1
    end
```

```
    Rational(misses, total)
  end

  def split_file(filepath)
    ext = File.extname(filepath)
    validation = File.open("./test/fixtures/validation#{ext}", "wb")
    training = File.open("./test/fixtures/training#{ext}", "wb")

    counter = 0
    File.open(filepath, 'rb').each_line do |l|
      if (counter) % 2 == 0
        validation.write(l)
      else
        training.write(l)
      end
      counter += 1
    end
    training.close
    validation.close
  end
end
```

在该测试中，我们从 –15 一直迭代到 15。这将覆盖我们能够遇到的大多数情况。待交叉验证结束后，我们可选择最优的 C，并将其运用于最终的模型。从技术上讲，这称为*网格搜索*（grid search），这种方法试图通过一组试验找到一个足够好的解。

现在我们需要来完善 `SentimentClassifier` 类的后台功能。我们在此利用 LibSVM 来构建模型，并生成一个很小的状态机：

```
# lib/sentiment_classifier.rb

class SentimentClassifier
  # build
  def initialize(corpus_set)
    @corpus_set = corpus_set
    @c = 2 ** 7
  end

  def c=(cc)
    @c = cc
    @model = nil
  end

  def words
    @corpus_set.words
  end

  def classify(string)
    if trained?
      prediction = @model.predict(@corpus_set.sparse_vector(string))
      present_answer(prediction)
    else
      @model = model
```

```
      classify(string)
    end
  end

  def trained?
    !!@model
  end

  def model
    puts 'starting to get sparse vectors'
    y_vec, x_mat = @corpus_set.to_sparse_vectors

    prob = Libsvm::Problem.new
    parameter = Libsvm::SvmParameter.new
    parameter.cache_size = 1000

    parameter.gamma = Rational(1, y_vec.length).to_f
    parameter.eps = 0.001

    parameter.c = @c
    parameter.kernel_type = Libsvm::KernelType::LINEAR

    prob.set_examples(y_vec, x_mat)
    Libsvm::Model.train(prob, parameter)
  end
end
```

这部分代码更加有趣，我们实际上是构建支持向量机来求解剩余的问题。如前所述，我们使用了非常标准的 LibSVM。我们首先构建自己的 `sparse_vectors`，然后加载一个新的 LibSVM 问题，最后将其参数设为默认值。

交叉验证完毕后，我们会看到最优的 C 为 128，其对应的错误率约为 30%。

6.4.6 随时间提升结果

要改进上述的 30% 的错误率，有多种策略，通常涉及一些试验：

- 删除停用词
- 改进符号化
- 使用不同的多项式核

你也可对同一组数据尝试几个别的算法，观察哪个性能更优。

6.5 小结

支持向量机算法非常适合求解两个可分类别的分类问题。我们可对该算法进行修改，使之能够处理多类分类问题，并避免像 K 近邻算法那样受到维数灾难的影响。本章介绍了如何运用 SVM 将忠诚用户和不忠诚用户分离开来，以及如何为影评数据赋予情绪标签。

第7章 神经网络

神经网络（或网络）可有效地以某个函数对观测数据进行映射。研究者已经能够利用神经网络来完成手写体检测、计算机视觉、语音识别等复杂任务，并取得了突破性的进展。

本质上，神经网络是一种从数据中学习的有效途径，并具有悠久的历史，可追溯到 19 世纪初。在本章中，我们将讨论神经网络算法的形成、发展、工作原理，以及一个基于字符频率的语言分类案例。



神经网络算法在函数逼近和有监督学习问题上具有突出的性能。它几乎适用于任何场合，且已被证实在实践中非常成功。然而，它只能对二值输入进行操作，并从复杂性和效率方面提出了一些挑战。

7.1 神经网络的历史

在神经网络最初被提出时，人们希望用它来研究人脑的工作机制。人脑中的神经元构成一种网络结构，并协同处理和理解输入和刺激。Alexander Bain 和 William James 均提出，人脑以能够处理大量信息的网状方式工作。这种神经元网络能够识别模式和从之前遇到的数据中进行学习。例如，当向一个孩子展示一张包含八条狗的照片后，他便开始理解狗的样子。

这项研究后来被扩展，将更加人工化的响应曲线包含进来，由 Warren McCulloch 和 Walter Pitts 发明了*阈值逻辑*（threshold logic）。阈值逻辑对二元信息进行整合，以确定逻辑真值。他们建议使用与阈值相关的*阶跃函数*，并以此确定是接受还是拒绝之前的信息之和。

经过多年研究，神经网络和阈值逻辑经过整合，形成了我们所熟知的人工神经网络。

7.2　何为人工神经网络

人工神经网络（参见图 7-1）是一种稳健函数，可接收任意输入并将其拟合到一组任意的二值输出。它在模糊匹配、构建稳健函数以与任何输出匹配方面具有突出的表现。实践中，神经网络被用于深度学习研究，以将图像与特征进行匹配等。

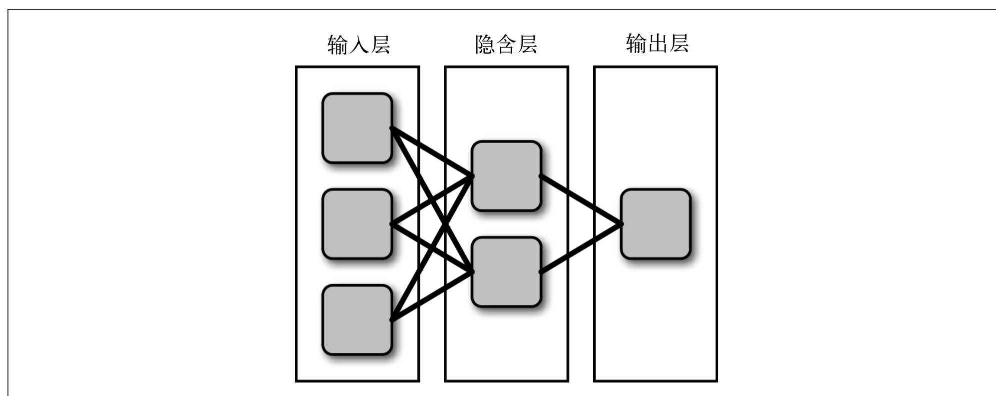


图 7-1：神经网络的可视化表示



模糊匹配是一种一般性的机器学习问题，其目标是依据之前的信息将输入与输出进行匹配。

神经网络的特别之处在于使用了位于隐含层的加权函数——神经元。利用神经元，可有效构建可逼近大量函数的网络。如果没有隐含层的函数，神经网络将只是一组简单的加权函数。

神经网络可用各层的神经元数目来表示。例如，如果某个网络的输入层有 20 个神经元，隐含层有 10 个神经元，而输出层有 5 个神经元，则该网络可表示为 20-10-5。如果隐含层的数目多于 1，则可将其表示为 20-7-7-5（中间的两个 7 表示该网络有两个隐含层，每层的结点数均为 7）。

简言之，神经网络由下列部件构成：

- 输入层
- 隐含层
- 神经元

- 输出层
- 训练算法

接下来将解释每个部件的含义及其工作原理。

7.2.1 输入层

输入层（如图 7-2 所示）是神经网络的入口点。它是给予以模型的输入的入口点。这一层中不含神经元，因为它主要用作隐含层的管道。输入类型十分重要，因为神经网络只能接收两种类型的输入：对称输入和标准输入。

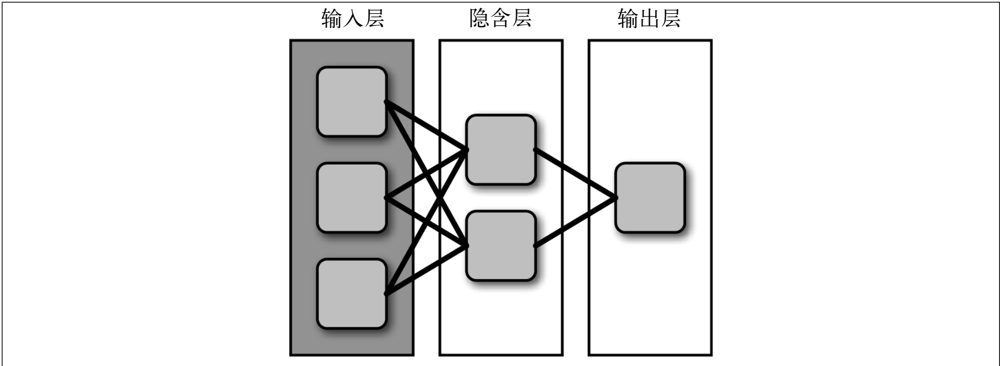


图 7-2：神经网络的输入层

训练神经网络时，我们拥有一些观测量和输入。以简单的异或运算（XOR）为例，真值表如表 7-1 所示。

表7-1：异或运算真值表

输入A	输入B	输出
F	F	F
F	T	T
T	F	T
T	T	F

在本例中，共有四个观测量和两个输入，其值非真即假。神经网络并不直接利用真或假，了解如何对输入编码才是最关键的。我们需要将它们变换为标准输入或对称输入。

1. 标准输入

输入值的标准范围是 0~1。在之前的异或例子中，我们可将真编码为 1，将假编码为 0。

这种类型的输入有一个缺陷：如果数据很稀疏，即绝大多数元素为 0，则可能会使结果产生偏倚。一个数据集拥有大量 0 意味着模型失效的风险也较高。仅当知道不存在稀疏数据时，才使用标准输入。

2. 对称输入

对称输入避免了存在大量 0 的问题。输入值的取值范围是 –1~+1。在之前的例子中，可将假取为 –1，而将真取为 +1。

这种类型的输入的好处是不会由于归零效应而使模型失效。此外，它不那么关注输入分布的中间值。如果为异或运算引入一个值“可能”，则可将其映射为 0 并忽略。

输入既可采用标准格式，也可采用对称格式，但使用时需要标注清楚，因为神经元输出的计算取决于输入的格式。

7.2.2 隐含层

如果没有隐含层，神经网络将只是一组加权线性组合。换言之，隐含层赋予了神经网络对非线性数据建模的能力（参见图 7-3）。

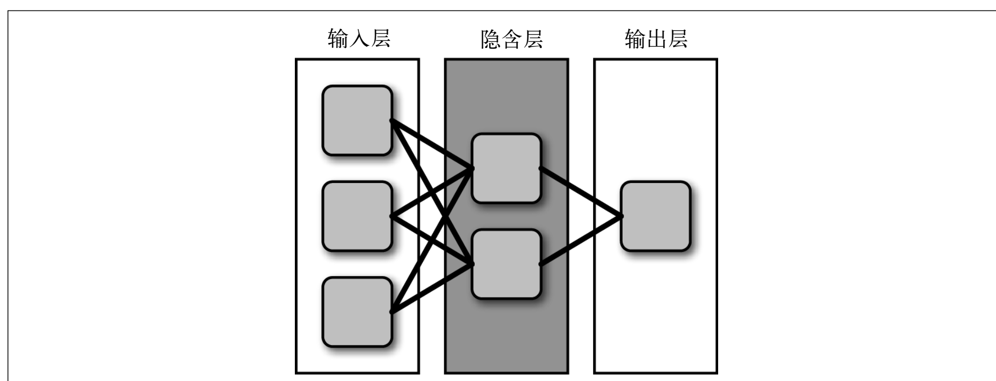


图 7-3：神经网络的隐含层

每个隐含层中都包含了一组神经元（如图 7-4 所示）。这些神经元将其输出直接传递给输出层。

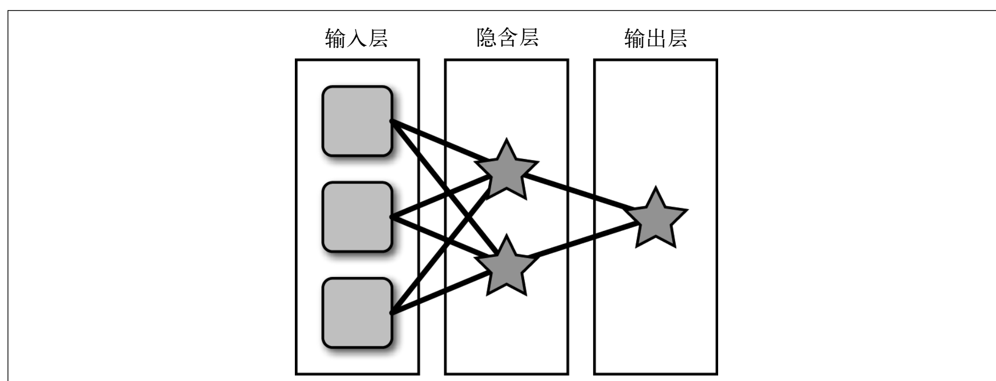


图 7-4：神经网络中的神经元

7.2.3 神经元

神经元完成的计算是将加权线性组合包裹进一个激活函数。加权线性组合（或和）是一种将前面的神经元的数据整合为一个输出，并作为下一层输入的方式。激活函数（如图 7-5 所示）是一种将数据归一化的方法，可使其格式变为对称型或标准型。

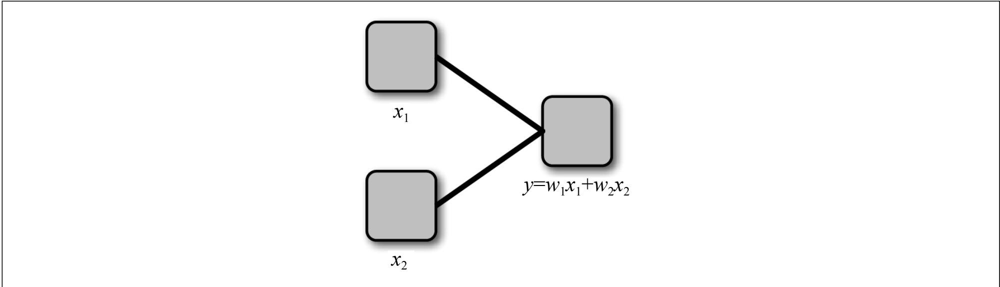


图 7-5：包裹进激活函数的神经元

当一个网络将信息前馈时，它会将之前的输入整合为一个加权和。我们取 y 的值，并依据激活函数计算激活值。

激活函数

如前所述，激活函数（表 7-2 列出了部分常用的激活函数）是一种将数据规一化为标准型或对称型的方法。它们均可微，且需要具有可微性，这是因为训练算法在学习权值时需要利用这一点。

表7-2：常见的激活函数

名称	标准型	对称型
Sigmoid	$\frac{1}{1+e^{-2\dot{s}um}}$	$\frac{2}{1+e^{-2\dot{s}um}}-1$
余弦函数	$\frac{\cos(sum)}{2}+0.5$	$\cos(sum)$
正弦函数	$\frac{\sin(sum)}{2}+0.5$	$\sin(sum)$
高斯函数	$\frac{1}{e^{sum^2}}$	$\frac{2}{e^{sum^2}}-1$
Elliott	$\frac{0.5\dot{s}um}{1+\lvert sum\rvert}+0.5$	$\frac{sum}{1+\lvert sum\rvert}$
线性函数	$sum>1?1:(sum<0:sum)$	$sum>1?1:(sum<-1?-1:sum)$
阈值函数	$sum<0?0:1$	$sum<0?-1:1$

使用激活函数最大的好处是它们提供了一种缓存每层输入值的途径。这是非常有用的，因

为神经网络拥有一种发现模式并忘记噪声的方法。

有两类主要的激活函数，一类是斜坡激活函数，另一类是周期激活函数。在大多数场合中，将斜坡激活函数（参见图 7-6 和图 7-8）作为默认选项都是适宜的。周期激活函数（参见图 7-7 和图 7-9）可用于随机数据，包括正弦和余弦函数。

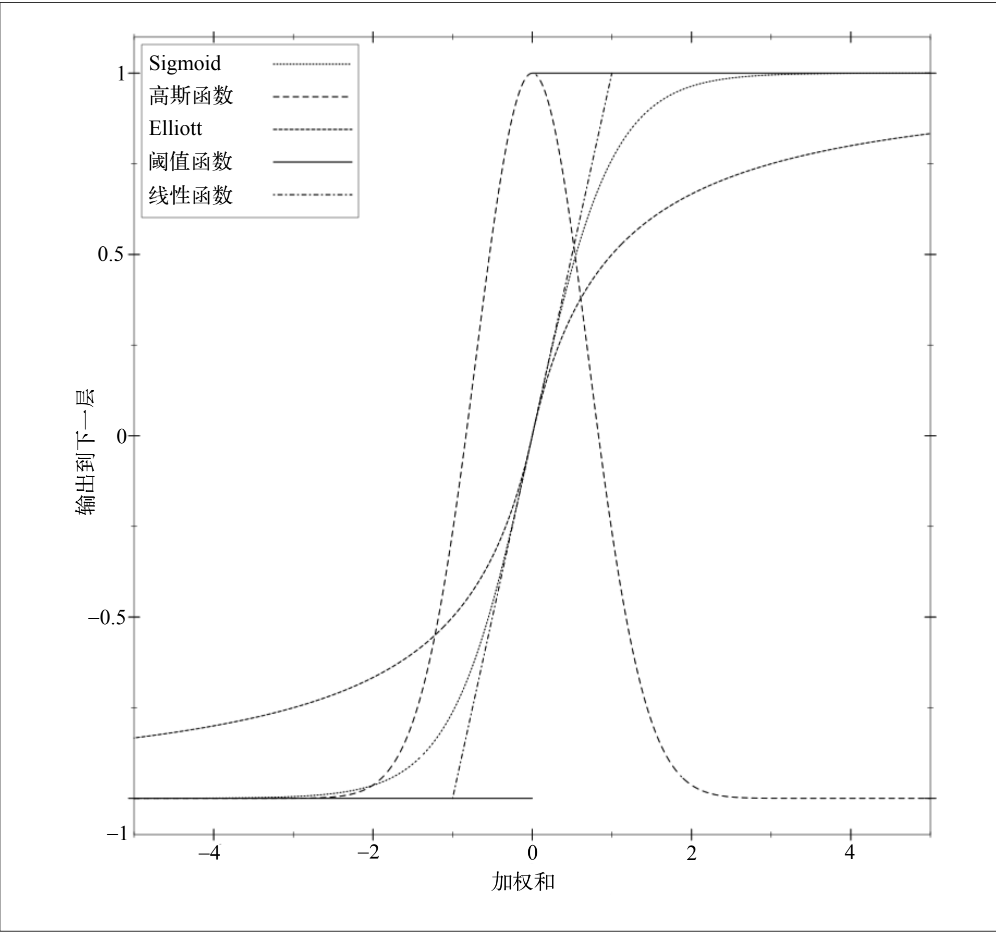


图 7-6：对称斜坡激活函数

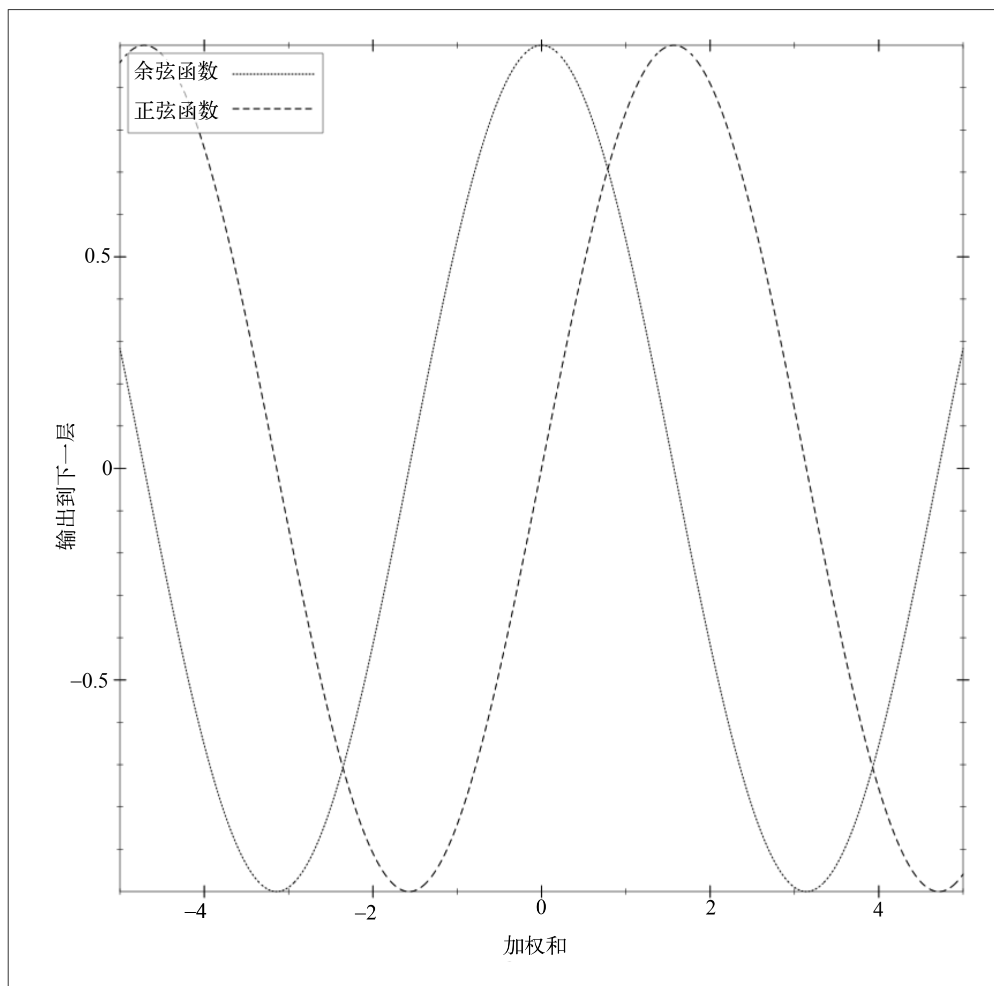


图 7-7：对称周期激活函数

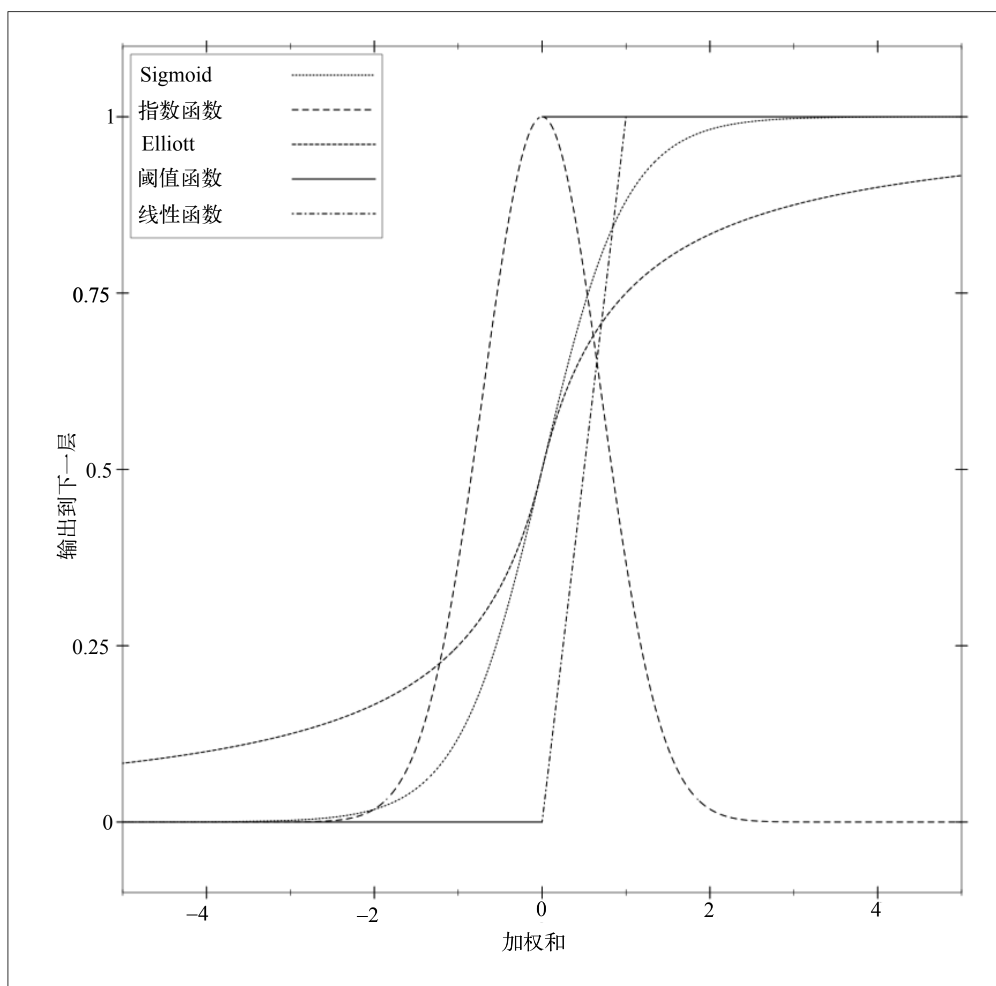


图 7-8：标准斜坡激活函数

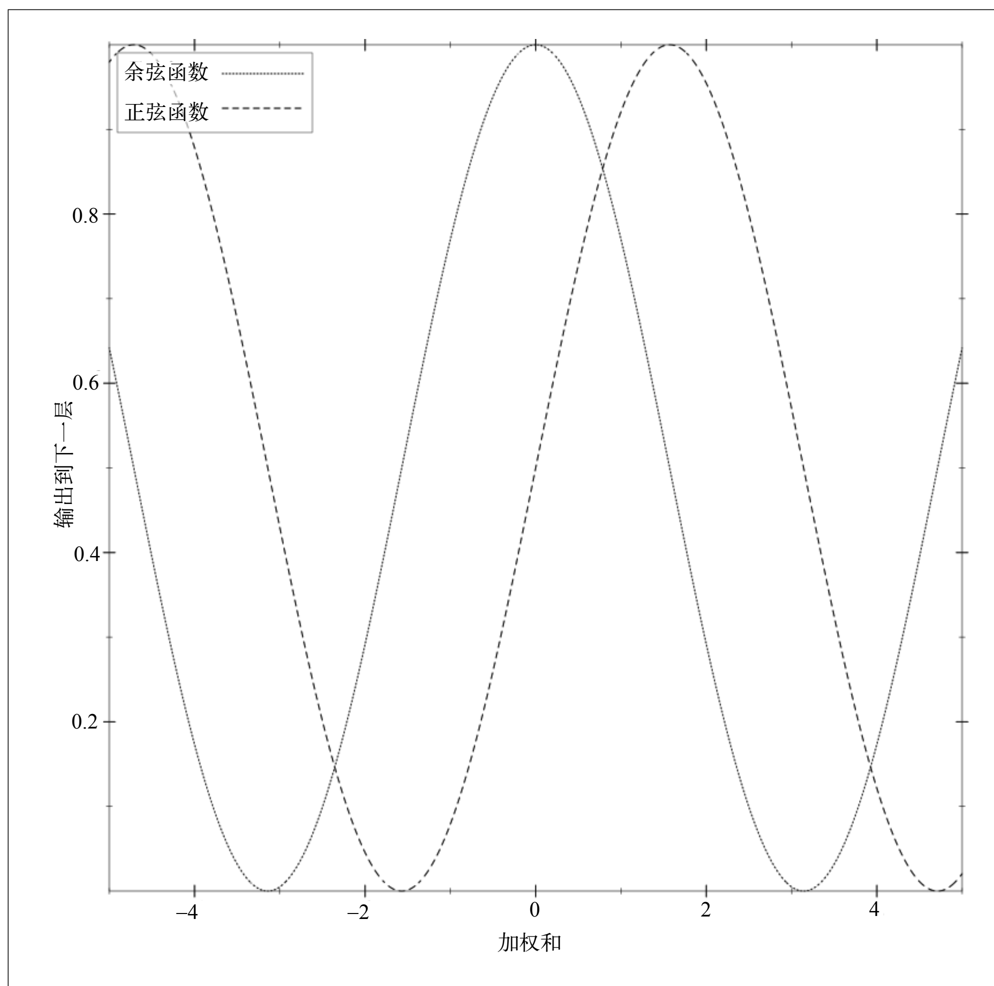


图 7-9：标准周期激活函数

由于 Sigmoid 具有平滑决策的能力，它是神经元使用的默认激活函数。Elloitt 是一种便于快速计算的 Sigmoid 型激活函数，因此我在本章案例中选择了它。当需要对一些看起来与某个随机过程关联的对象进行映射时，可使用正弦函数和余弦函数。在大多数情况下，这些三角函数对于我们都用处不大。

所有的运算均在神经元中完成。神经元接收前一层输入的加权和，送入一个激活函数，并将结果钳位在 [0,1] 或 [–1,+1] 内。对于拥有两个输入的神经元，其输出可表示为 $y = \phi(w_1\dot{x}_1 + w_2\dot{x}_2)$，其中 ϕ 是一个激活函数（如 Sigmoid），w_i 是由训练算法所确定的权值。

7.2.4 输出层

输出层与输入层类似，只是它包含了一组神经元。它是数据的模型出口。与输入层类似，输出层的输入类型也分为对称型和标准型。

输出层确定了有多少神经元产生输出，它代表了我们所要建模的函数（参见图 7-10）。对于一个其输出值表示交通灯状态（红、绿、黄）的函数，我们拥有三个输出（每种颜色对应一个输出）。这些输出中的每一个都包含了我们所希望的逼近结果。

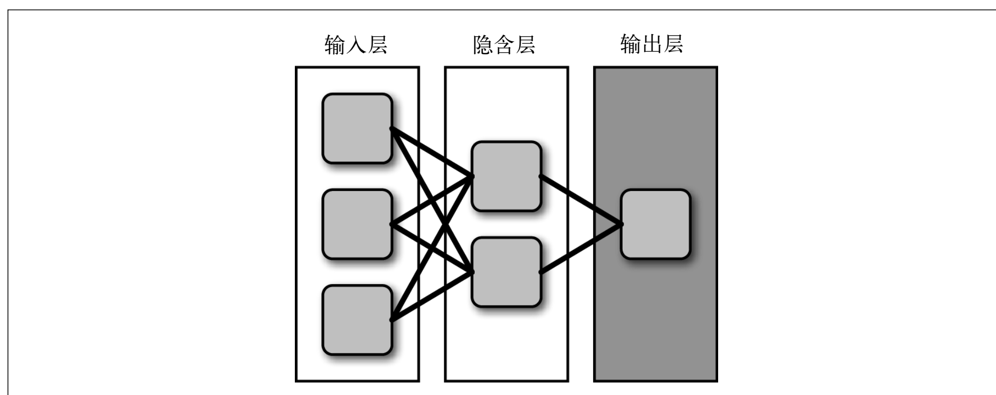


图 7-10：神经网络的输出层

7.2.5 训练算法

如前所述，每个神经元的权值来自于一个训练算法。训练算法有很多种，但最常见的是：

- 反向传播（BP）算法
- QuickProp
- RProp

所有这些算法都可为每个神经元找到最优的权值。它们通过*迭代*（也称 epoch）来达到这个目标。对于每个 epoch，训练算法都会经过整个神经网络，并将实际输出与期望输出进行比较。学习算法无一不是从过去的错误计算中进行学习的。

这些算法有一个共同点：它们均是试图在一个*凸误差曲面*内寻找最优解。可将凸误差曲面想象为一个碗，其中包含一个最小值。设想你最初位于一座山的顶部，并希望到达山谷的谷底，但山谷中密布了很多树木。你看不清前方的路，但清楚自己希望到达山谷的谷底。你会依据局部输入来行进，并猜测下一步走到哪里。这便是梯度下降算法的基本原理（即沿着山谷向下行走，以将误差最小化），如图 7-11 所示。训练算法也是一样，利用局部信息来将误差最小化。

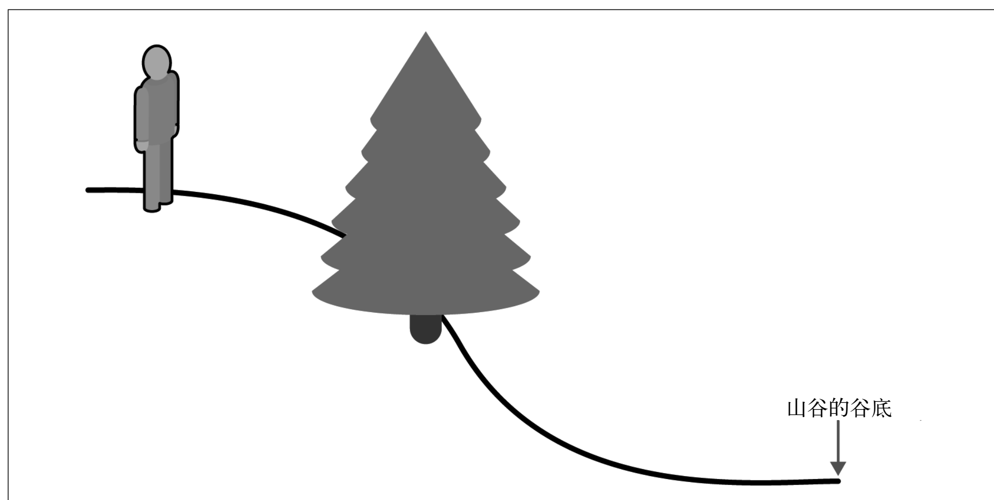


图 7-11：梯度下降算法示意

1. Delta规则

虽然我们可以求解一个大规模方程组，但采用迭代策略效果更胜一筹。我们并不试图去计算误差函数对权值的偏导，而是计算每个神经元权值的变化量。这被称为 Delta 规则，如下所示：

$$\delta wji = \alpha\big(tj - \phi(hj)\big)\phi'(hj)xi$$

该规则表明神经元 j 的第 i 个权值的增量为：

```
alpha * (expected - calculated) * derivative_of_calculated * input_at_i
```

其中，`alpha` 为学习率（learning rate），其值很小。著名的反向传播算法（Delta 规则的一般情形）正是基于这种思想。

2. 反向传播

反向传播算法是用于确定神经元权值的三种算法中最简单的一种。我们将误差定义为 $(\text{expected} * \text{actual})^2$，其中 expected 是期望输出，而 actual 为神经元的实际输出。我们希望找到使导数为 0 的位置，即极小点：

$$\Delta w(t) = -\alpha(t - y)\phi' x_i + \epsilon \Delta w(t - 1)$$

其中，ϵ 为动量因子（momentum factor），它会将之前的权值变化转化为当前的权值增量，其中 α 为学习率。

反向传播算法的不足在于它需要许多 epoch 才能收敛。直到 1988 年，研究者们才有能力训练简单的神经网络。人们对提升效率的研究导致了一种全新的学习算法的产生——QuickProp。

3. QuickProp

Scott Fahlman 在研究了如何改进反向传播算法后，提出了 QuickProp 算法。他声称反向传播算法需要很长时间才能收敛。他建议我们选择最大的步长，而无需担心越过最优解。

Fahlman 认为存在两种改进反向传播算法的方式：增加动量项并动态调整学习率，以及利用误差相对于每个权值的二阶导数。对于第一种情形，我们可更好地优化每个权值；在第二种情形中，我们可利用牛顿的函数逼近法。

QuickProp 与反向传播算法的主要区别在于，在上一 epoch 期间，我们保存了所计算的误差导数，以及权值的当前权值和先前权值之间的差异。

为计算时刻 t 的权值变化，可使用下列公式：

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \dot{\Delta} w(t-1)$$

该公式有将权值改变过多的风险，因此对于最大增量有一个专门的新参数。任何权值都不允许在数量上大于最大增长率与权值在上一步增量之积。

4. RProp

由于 RProp 的快速收敛性，它已成为最常用的学习算法。该算法由 Martin Riedmiller 于 20 世纪 90 年代提出，并在后来进行了若干改进。由于它洞悉到算法可在一个 epoch 内多次更新权值，因此它可快速收敛到某个解。该算法并不依据某个公式去计算权值的增量，而是仅利用增量的符号以及一个提升因子和下降因子。

为了解该算法的实现细节，我们需要定义少量常量（或默认值）。有一种方式可确保该算法不会永远计算下去或产生震荡。这些默认值都取自 FANN 库。

这个基本算法用 Ruby 语言更易于解释，而无需写出偏导。



为降低阅读的难度，请注意我并未计算误差的梯度（即误差相对于某个特定权值的变化）。

下列通过一段纯 Ruby 代码帮助你理解 RProp 算法：

```
neurons = 3
inputs = 4

delta_zero = 0.1
increase_factor = 1.2
decrease_factor = 0.5
```

```ruby
delta_max = 50.0
delta_min = 0
max_epoch = 100
deltas = Array.new(inputs) { Array.new(neurons) { delta_zero }}
last_gradient = Array.new(inputs) { Array.new(neurons) { 0.0 } }


sign = ->(x) {
  if x > 0
    1
  elsif x < 0
    -1
  else
    0
  end
}

weights = inputs.times.map {|i| rand(-1.0..1.0) }

1.upto(max_epoch) do |j|
  weights.each_with_index do |i, weight|
    # 当前梯度从每层中每个值的变化推导而来
    gradient_momentum = last_gradient[i][j] * current_gradient[i][j]

    if gradient_momentum > 0
      deltas[i][j] = [deltas[i][j] * increase_factor, delta_max].min
      change_weight = -sign.(current_gradient[i][j]) * deltas[i][j]
      weights[i] = weight + change_weight
      last_gradient[i][j] = current_gradient[i][j]
    elsif gradient_momentum < 0
      deltas[i][j] = [deltas[i][j] * decrease_factor, delta_min].max
      last_gradient[i][j] = 0
    else
      change_weight = -sign.(current_gradient[i][j]) * deltas[i][j]
      weights[i] = weights[i] + change_weight
      last_gradient[i][j] = current_gradient[i][j]
    end
  end
end
```

为构建一个神经网络，这些都是你需要理解的基础内容。接下来，我们将讨论如何构建神经网络，以及为使其更加有效必须做出哪些决策。

7.3 构建神经网络

在开始构建神经网络之前，必须先回答下列问题：

- 应使用多少个隐含层？
- 每层的神经元数目应为多少？
- 误差容限应设为多大？

7.3.1　隐含层数目的选择

前面我们介绍过，神经网络的独特之处在于隐含层的使用。如果将隐含层移除，神经网络将只能完成简单的线性运算。你一定不愿随意选用隐含层的数目，不过下列三种试探法会对你有所帮助。

- 隐含层的数目不宜超过 2；否则，发生过拟合的几率便会增大。如果层数过多，网络就会开始记忆训练数据。显然，这与我们发现模式的初衷相悖。
- 一个隐含层所完成的实际上是一个连续映射。这是最常见的情形。大多数神经网络只包含一个隐含层。
- 两个隐含层将能够实现任意连续映射。这是一种不太常见的情形，但如果你了解自己缺少连续映射，便可使用两个隐含层。

对于隐含层数目的选择，没有什么固定规则可循。归纳起来，无非是努力降低对数据过拟合或欠拟合的风险。

7.3.2　每层中神经元数目的选择

神经网络是性能优异的整合器和无与伦比的扩展器。神经元自身的输入是位于其前的神经元输出的加权和，因此其扩张能力通常略逊于整合能力。例如，隐含层共有 2 个结点，而输出层有 30 个结点，则对于每个输出神经元，都拥有两个输入。因而缺乏足够的熵或数据来形成一个拟合度较高的模型。

强调聚合而非扩张，可得到下列启发式策略。

- 隐含层神经元的数目应当介于输入的个数和输出层神经元数目之间。
- 隐含层神经元的数目应为输入层尺寸的 2/3 加上输出层的尺寸。
- 隐含层神经元的数目应当小于输入层尺寸的两倍。

总之，我们需要通过反复试验来确隐含层神经元的数目，因为它会影响模型的交叉验证性能以及收敛性。这只是一个起点。

7.3.3　误差容限和最大epoch的选择

误差容限决定了训练何时停止。我们永远无法得到一个完美的解，而只能收敛到某个解。如果希望拥有一个性能良好的算法，则错误率应当具有较低的水平，如 0.01%。但在大多数情况下，由于算法对误差的容忍度较低，训练时长通常都较久。

很多人最初会将误差容限设为 1%。通过交叉验证，这个指标可能会进一步降低。用神经网络的术语来说，误差容限是内部的，用均方误差来度量，并为网络定义了一个训练停止点。

神经网络需要通过多个 epoch 进行训练，在训练开始之前便需要对最大 epoch 进行设置。如果算法经过 10 000 轮迭代后得到了一个解，则该网络被过度训练的风险便很高，从而使网络灵敏度过高。在训练阶段，一个好的起点是将最大 epoch 数取为 1000。这样，既可对某种程度上的复杂性进行建模，同时也不会训练过度。

最大 epoch 数和最大误差都决定了收敛点。它们都可指示训练算法何时可以终止并生成最终的神经网络。至此，我们对神经网络已有了足够的了解，下面我们通过一个实际案例来加深理解。

7.4　利用神经网络对语言分类

一门语言中所使用的字符与该语言自身有直接的关系。依据其字符，普通话是可以识别的，因为每个字符都对应一个特定的词。拉丁语系中的很多语言也是如此，但需要依据字母频率来判定。

如果观察英语句子“The quick brown fox jumped over the lazy dog”及其德语版本“Der schnelle braune Fuchs sprang über den faulen Hund”，可得到如表 7-3 所示的词频数据。

表7-3：英语和德语句子之间的词频差异

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	ü
英语	1	1	1	2	4	1	1	2	1	1	1	1	1	1	4	1	1	2	0	2	2	1	1	1	1	1	0
德语	3	2	2	3	7	2	1	3	0	0	0	3	0	6	0	1	0	4	2	0	4	0	0	0	0	1	1
差	2	1	1	1	3	1	0	1	1	1	1	2	1	5	4	0	1	2	2	2	2	1	1	1	1	0	1

德语和英语之间存在微妙的差异。德语中使用 N 较多，而英语中使用 O 较多。那么如何将此扩展到更多的欧洲语言呢？具体说来，如何构建一个模型来对用英语、波兰语、德语、芬兰语、瑞典语或挪威语书写的句子进行分类？

本例中，我们将构建一个依据句子中字符频率来预测语言的简单模型。在开始之前，我们需要一些数据。为此，我们打算使用世界上译本最多的书籍——《圣经》。我们抽取了 Matthew 和 Acts 中的全部章节。

我们将要采取的方法是从这些文本文件中提取所有的句子，并创建一些其分量归一化到 0~1 的频率向量。据此，我们将训练一个可接收这些输入，并将其映射为一个 6 维向量的神经网络。这个向量中仅有与输入对应的语言的索引所对应的分量为 1，而其余分量为 0。例如，若我们用于训练的语言索引是 3，则该向量应当为 [0,0,0,1,0,0]（下标从 0 算起）。



安装说明

本例中使用的所有代码均可从 GitHub（https://github.com/thoughtfulml/examples/tree/master/6-neural-networks）获取。

由于 Ruby 处于持续的更新和变化之中，因此要想获悉如何快速上手这些例子，README 文件无疑是最佳选择。

数据获取

如果希望抓取数据，我编写了下列脚本来帮助你从 Biblegateway.com 抓取《圣经》中的句子：

```
require 'nokogiri'
require 'open-uri'

url = "http://www.biblegateway.com/passage/"

languages = {
  'English' => {
    'version' => 'ESV',
    'search' => ['Matthew', 'Acts']
  },
  'Polish' => {
    'version' => 'SZ-PL',
    'search' => ['Ewangelia+wedlug+św.+Mateusza', 'Dzieje+Apostolskie']
  },
  'German' => {
    'version' => 'HOF',
    'search' => ['Matthaeus', 'Apostelgeschichte']
  },
  'Finnish' => {
    'version' => 'R1933',
    'search' => ['Matteuksen', 'Teot']
  },
  'Swedish' => {
    'version' => 'SVL',
    'search' => ['Matteus', 'Apostlagärningarna']
  },
  'Norwegian' => {
    'version' => 'DNB1930',
    'search' => ['Matteus', 'Apostlenes-gjerninge']
  }
}

languages.each do |language, search_pattern|
  text = ''

  search_pattern['search'].each_with_index do |search, i|
    1.upto(28).each do |page|
      puts "Querying #{language} #{search} chapter #{page}"
```

```
        uri = [
          url,
          URI.encode_www_form({
            search: "#{URI.escape(search)}+#{page}",
            version: "#{search_pattern.fetch('version')}"
          })
        ].join('?')
        puts uri
        doc = Nokogiri::HTML.parse(open(uri))
        doc.css('.passage p').each do |verse|
          text += verse.inner_text.downcase.gsub(/[\d,;:\\\-\"]/,'')
        end
      end
      File.open("#{language}_#{i}.txt", 'wb') {|f| f.write(text)}
    end
  end
```

这段程序可下载并保存 Acts 和 Matthew 中诗句的多语言版本。你可尽情地去尝试更多的语言版本！

7.4.1　为语言编写接缝测试

为读取训练数据，我们需要构建一个类来解析输入，并作为神经网络的接口。为此，我们将这个类的名称定为 Language。Language 类的目的是读取一个用某种语言编写的文本文件，并求出其字符频率的分布。必要时，Language 类将输出一个这些字符的频率向量，该向量各分量之和为 1。我们的全部输入都介于 0~1 之间。参数包括：

- 我们希望确保数据是正确的，且和为 1；
- 我们希望数据中不包含 UTF-8 编码的空格或标点符号；
- 我们希望所有的字符均为小写。A 应转换为 a，而 Ä 应转换为 ä。

这将有助于保证输入为文本文件、输出为散列数组的 Language 类的正确性：

```
# encoding: utf-8
# test/lib/language_spec.rb

require ''spec_helper'
require 'stringio'


describe Language do
  let(:language_data) {
    <<-EOL
    abcdefghijklmnopqrstuvwxyz.
    ABCDEFGHIJKLMNOPQRSTUVWXYZ.
    \u00A0.
    !~@#$\%^&*()_\+'?[]""''——<>»«›‹–„/.
    ïëéüòèöÄÖßÜøæåÅØóąłżŻśęńŚćźŁ.
```

```
    EOL
  }

  let(:special_characters) { language_data.split("\n").last.strip }

  let(:language_io) { StringIO.new(language_data) }

  let(:language) { Language.new(language_io, 'English') }

  it 'has the proper keys for each vector' do
    language.vectors.first.keys.must_equal ('a'..'z').to_a
    language.vectors[1].keys.must_equal ('a'..'z').to_a

    special_chars = "ïëéüòèöäößüøæååØóąłżżśęńśćź".split(//).uniq.sort

    language.vectors.last.keys.sort.must_equal special_chars
  end

  it 'sums to 1 for all vectors' do
    language.vectors.each do |vector|
      vector.values.inject(&:+).must_equal 1
    end
  end

  it 'returns characters that is a unique set of characters used' do
    chars = ('a'..'z').to_a
    chars.concat "ïëéüòèöäößüøæååØóąłżżśęńśćź".split(//).uniq

    language.characters.to_a.sort.must_equal chars.sort
  end
end
```

此时，我们尚未开始实际编写 Language 类，因此所有的测试均会失效。对于第一个目标，我们需要统计所有字母字符的出现次数，并停留在某个句子上。完成此功能的代码大致如下：

```
# encoding: utf-8

# lib/language.rb

class Language
  attr_reader :name, :characters, :vectors
  def initialize(language_io, name)
    @name = name
    @vectors, @characters = Tokenizer.tokenize(language_io)
  end
end

# lib/tokenizer.rb

module Tokenizer
  extend self
  PUNCTUATION = %w[~ @ # $ % ^ & * ( ) _ + ' [ ] " " ' ' — < > » « › ‹ - „ /]
```

```
  SPACES = [" ", "\u00A0", "\n"]
  STOP_CHARACTERS = ['.', '?', '!']

  def tokenize(blob)
    unless blob.respond_to?(:each_char)
      raise 'Must implement each_char on blob'
    end

    vectors = []
    dist = Hash.new(0)

    characters = Set.new
    blob.each_char do |char|
      if STOP_CHARACTERS.include?(char)
        vectors << normalize(dist) unless dist.empty?
        dist = Hash.new(0)
      elsif SPACES.include?(char) || PUNCTUATION.include?(char)

      else
        character = char.downcase.tr("ÅÄÜÖËÏŚŻŁ", "åäüöëïśźł")
        characters << character
        dist[character] += 1
      end
    end
    vectors << normalize(dist) unless dist.empty?

    [vectors, characters]
  end
end
```

现在，我们已经拥有了部分能够工作的代码。此外还有一些有趣的地方值得我们注意。首先，像 Ä 这样的特殊字符没有被转为小写 ä。为此，必须使用 `String#tr`；其次，存在 Unicode 空格，它可表示为 `\u00a0`。

至此，我们遇到了一个新问题，即所有数据之和并不为 1。我们将引入一个新的函数 `nomralize`，该函数接收一个散列值，并将 `x/sum(x)` 运用于所有的值。请注意，我使用了 `Rational`，这是 Ruby 1.9.x 中的新特性，它能够增强计算的可靠性，且只在必要时方进行浮点运算：

```
# lib/tokenizer.rb

module Tokenizer
  # 符号化

  def normalize(hash)
    sum = hash.values.inject(&:+)
    Hash[
      hash.map do |k,v|
        [k, Rational(v,  sum)]
      end
    ]
```

```
  end
end
```

至此，Language 类已圆满实现。对于作为神经网络接口的类，我们已经介绍了完整的测试。现在我们开始构建 Network 类。

7.4.2 网络类的交叉验证

我从《圣经》中获取用于语言分类的训练数据，因为它是史上译本最多的著作。具体而言，我决定下载 Matthew 和 Acts 的英语版、芬兰语版、德语版、挪威语版、波兰语版以及瑞典语版。来自 Acts 和 Matthew 的文本形成了数据集的一种自然划分，我们对从 Acts 训练得到的模型定义了 12 项测试，并将它与 Matthew 数据的结果对比，然后反过来再次观察。

代码如下：

```
# test/cross_validation_spec.rb
# encoding: utf-8

require 'spec_helper'

# 该散列对网络进行了缓存,以加速测试过程,这一点很重要

networks = {}

describe Network do
  def language_name(text_file)
    File.basename(text_file, '.txt').split('_').first
  end

  def compare(network, text_file)
    misses = 0.0
    hits = 0.0

    file = File.read(text_file)

    file.split(/[\.!\?]/).each do |sentence|
      sentence_name = network.run(sentence).name

      if sentence_name == language_name(text_file)
        hits += 1
      else
        misses += 1
      end
    end

    total = misses + hits

    assert(
      misses < (0.05 * total),
```

```
      "#{text_file} has failed with a miss rate of #{misses / total}"
    )
  end

  def load_glob(glob)
    Dir[File.expand_path(glob, __FILE__)].map do |m|
      Language.new(File.open(m, 'r+'), language_name(m))
    end
  end

  let(:matthew_languages) { load_glob('../../data/*_0.txt') }
  let(:acts_languages) { load_glob('../../data/*_1.txt') }

  let(:matthew_verses) {
    networks[:matthew] ||= Network.new(matthew_languages).train!
    networks[:matthew]
  }

  let(:acts_verses) {
    networks[:acts] ||= Network.new(acts_languages).train!
    networks[:acts]
  }

  %w[English Finnish German Norwegian Polish Swedish].each do |lang|
    it "Trains and cross-validates with an error of 5% for #{lang}" do
      compare(matthew_verses, "./data/#{lang}_1.txt")
      compare(acts_verses, "./data/#{lang}_0.txt")
    end
  end
end
```

在这个项目的根目录中有一个名为 data 的文件夹，其中包含了形式为 `Language_0.txt` 和 `Language_1.txt` 的文件，其中 `Language` 为语言名称，`_0` 为映射到 Matthew 的索引，而 `_1` 为映射到 Acts 的索引。



训练神经网络需要一定的时间，因此我只训练了两个网络模型，一个从 Acts 的章节中习得，而另一个从 Matthew 的章节中习得。此时，我们定义了 12 项测试。当然，由于我们尚未编写 `Network` 类，因此训练还无从谈起。为实现 `Network` 类，我们先定义一个能够接收 `Language` 类的数组的初始版本。其次，由于我们不希望手工编写所有的神经网络训练代码，所以我们打算使用一个名为 Ruby-Fann 的库，该库提供了 FANN 的接口。我们最初的主要目标是获得一个能够接收输入并进行训练的神经网络。

对之前的测试进行填充，我们可用如下方式来构建网络对象：

```
# lib/network.rb

require 'ruby-fann'
```

```
class Network
  def initialize(languages, error = 0.005)
    @languages = languages
    @inputs = @languages.map {|l| l.characters.to_a }.flatten.uniq.sort
    @fann = :not_ran
    @trainer = :not_trained
    @error = error
  end

  def train!
    build_trainer!
    build_standard_fann!
    @fann.train_on_data(@trainer, 1000, 10, @error)
    self
  end

  def code(vector)
    return [] if vector.nil?
    @inputs.map do |i|
      vector.fetch(i, 0.0)
    end
  end

  private
  def build_trainer!
    payload = {
      :inputs => [],
      :desired_outputs => []
    }

    @languages.each_with_index do |language, index|
      inputs = []
      desired_outputs = [0] * @languages.length
      desired_outputs[index] = 1

      language.vectors.each do |vector|
        inputs << code(vector)
      end

      payload[:inputs].concat(inputs)

      language.vectors.length.times do
        payload[:desired_outputs] << desired_outputs
      end
    end

    @trainer = RubyFann::TrainData.new(payload)
  end

  def build_standard_fann!
    hidden_neurons = (2 * (@inputs.length + @languages.length)) / 3

    @fann = RubyFann::Standard.new(
      :num_inputs => @inputs.length,
      :hidden_neurons => [ hidden_neurons ],
```

```
      :num_outputs => @languages.length
    )

    # 注意,库将Elliott拼写错了
    @fann.set_activation_function_hidden(:elliot)
  end
end
```

既然我们已经拥有了合适的输入和输出，模型已经建立，我们应当运行完整的 cross_validation_test.rb。当然，还存在一个错误，因为我们无法将新输入送入该网络。为解决这个问题，我们需要构建一个名为 `#run` 的函数。至此，我们的代码已经可以工作，如下所示：

```
# lib/network.rb

require 'ruby-fann'
class Network
  # initialize
  # train!
  # code

  def run(sentence)
    if @trainer == :not_trained || @fann == :not_ran
      raise 'Must train first call method train!'
    else
      vectors, characters = Tokenizer.tokenize(sentence)
      output_vector = @fann.run(code(vectors.first))
      @languages[output_vector.index(output_vector.max)]
    end
  end
end
```

从这里开始，事情变得有趣起来，我们发现德语、英语、瑞典语以及挪威语都无法通过测试。由于我们的代码已经可以工作，现在我们开始依据单元测试来对神经网络进行调整。

虽然我们将标准设定得较高，但可通过调校网络的参数来达到。

7.4.3 神经网络的参数调校

此时，我们已将激活函数改为 Elliott 函数，除挪威语、瑞典语和德语外的结果都得到了提升。英语的正确率为 100%，但 epoch 数略为增加。将内部错误率折半，降至 0.005 是我们下一步要做的事情，这可通过将 `@fann.train_on_data` 的最后一个参数设为 0.005 来实现。最终，我们将实现预设的目标。

进一步的参数调校留给你作为练习。你可尝试不同的激活函数，以及内部衰减率或误差。由此我们得到启发：通过初始测试来测定准确率，可尝试多种方法。

7.4.4 收敛性测试

在继续之前，可重设神经网络类的最大 epoch 数，使其比你看到的高出 20%~50%，以确保目标函数的下降趋于平稳。在我们的例子中，我看到模型训练需要花费大约 200 个 epoch，因此我将最大 epoch 数重设为 300。

7.4.5 神经网络的精度和查全率

我们再进一步，当我们在产品环境中部署这段神经网络代码时，需要引入精度和查全率这两种度量，并追踪其随时间的变化，以形成信息回路。这些度量将依据用户输入来计算。

我们可通过在用户接口询问预测是否正确，来测量精度和查全率。从文本中，我们可以捕捉错误的和正确的分类，并在下次训练时反馈给我们的模型。

关于精度和查全率检测的更多细节，请参阅第 9 章。

在产品环境中，我们需要检测的神经网络性能指标包括分类的次数以及出错的次数。

7.4.6 案例总结

神经网络算法是一种通过迭代实现信息映射和学习的有趣方式，对于将句子映射为语言这个示例而言，它有很好的表现。将这段代码加载到一个 IRB 会话中，当我输入“meep moop”时，模型会将其分类为挪威语！请自由地测试这些代码。

7.5 小结

神经网络算法是机器学习工具集中一种十分强大的工具。神经网络是一种通过函数模型对之前的观测进行映射的方式。虽然它们常被用作黑箱模型，但只要通过一点点数学推导和图示还是可以理解的。你可利用神经网络还完成许多任务，如将字母频率映射为语言，或手写体检测。有很多问题都可用这种算法来求解，而且关于这个话题的比较深入的相关书籍也越来越多。任何由 Geoffrey Hinton 撰写的著作或文章都值得一读，比如 *Unsupervised Learning: Foundations of Neural Computation*（computaitonal Neuroscience）。

本章介绍了作为人脑的人工版本的神经网络，并解释了它们通过加权函数对输入进行汇总的工作原理。之后，这些加权函数的输出会在一定范围内归一化。许多算法都可用于训练这些权值，但最流行的当属 RProp 算法。最后，我们通过一个将句子映射为语言的示例对上述内容进行了总结。

第 8 章
聚类

如果你曾经去过图书馆，想必已经接触过聚类。*杜威十进分类法*（Dewey Decimal System）便是聚类的一种形式。杜威所构建的这个分类法用大量粒度逐渐细化的阿拉伯数字作为标记符号，开创了文献分类法的新纪元。

将数据点或书籍分组，对于信息组织很有用。由于我们不清楚图书应当归属的具体类别，因此只希望将它们划分为若干不同的类别。

这种类型的问题与我们之前遇到的大不相同。到目前为止，我们接触的全部问题都试图找出最优的函数逼近，以将数据分配到某个数据集和标签。现在，我们更关心数据本身，而非标签。

通过本章的学习，你将了解到，聚类也有一个缺陷，即它不像其他算法那样兼顾多种不变性。这称作*不可能性定理*（impossibility theorem）。

考虑到其应用的广泛性，本章将先对聚类展开一般性的讨论，然后介绍 *K* 均值聚类算法和期望最大化（Expectation Maximization，EM）聚类算法。本章最后将给出一个将爵士乐按曲式分组的示例。



聚类算法是一种无监督学习问题，在数据分组方面有突出的表现。当然，在实际运用时，这类算法也会出现一些问题，即会受到不可能性定理的制约。

8.1　用户组

对于许多业务和市场营销而言，将人们划分到不同的组（簇）中非常有意义。例如，你的第一名客户与你的第一万名客户或第一百万名客户不同。将不同的用户定义为不同的组，这一过程十分常见。如果我们打算依据行为或注册时间将客户有效地划分到不同的组中，则可通过将营销策略多样化来为他们提供更优质的服务。

但这类问题的难点在于，我们缺少客户组的预定义标签。为解决这个问题，你可查看每个人在何年何月成为客户。但这里所做的假设是，首次购买的时间是对用户进行分组的决定性因素。如果首次购买时间与客户是否属于某一组无关，应如何处理？例如，他们可能仅有一次购物行为，或者此后进行了多次购买。

也可依据我们对用户的了解来对其分组。例如，我们知道他们的注册时间、所花费的金额以及最喜欢的颜色。在过去两年中，仅有 10 名用户进行了注册（希望你在过去两年中的注册用户多于此，这里做此假设是为了简化问题）。表 8-1 展示了我们在过去两年中收集到的用户数据。

表8-1：过去两年间收集到的用户数据

用户ID	注册时间	花销	最喜欢的颜色
1	Jan 14	$40	N/A
2	Nov 3	$50	橙色
3	Jan 30	$53	绿色
4	Oct 3	$100	品红
5	Feb 1	$0	蓝绿色
6	Dec 31	$0	紫色
7	Sep 3	$0	浅紫色
8	Dec 31	$0	黄色
9	Jan 13	$14	蓝色
10	Jan 1	$50	米黄色

给定这些数据，我们希望定义一个从每个用户到一个组的映射。观察该表的各行会发现，最喜欢的颜色属于不相关数据。这一信息无法提供一种有意义的方式来对用户分组。因此，我们只能利用花销和注册时间两项内容。看起来，有一组用户产生了开销，而另一组没有。在注册时间一列中，你会发现有很多用户在年初和年末以及九月、十月或十一月注册。

现在，我们需要确定要生成的簇的数目。由于我们的数据集规模很小，因此我们只将其分为两组。这意味着我们可将用户分为如表 8-2 所示的两个组。

表8-2：对原始数据集手工分组的结果

用户ID	注册时间（距离1月1日的天数）	花销	分组号
1	Jan 14 (13)	$40	1
2	Nov 3 (59)	$50	2
3	Jan 30 (29)	$53	1
4	Oct 3 (90)	$100	2
5	Feb 1 (31)	$0	1
6	Dec 31 (1)	$0	1
7	Sep 3 (120)	$0	2
8	Dec 31 (1)	$0	1
9	Jan 13 (12)	$14	1
10	Jan 1 (0)	$50	1

我们已将用户分为两组：第一组中包含了 7 名客户，我们可命名为“年初注册”组；第二组中则包含了另外 3 名客户。

那么，如何用算法化的方法来完成该任务呢？下面几节将介绍何为 *K* 均值聚类以及 EM 聚类算法的理论意义。本章最后将给出一个唱片分类的示例。

8.2　*K*均值聚类

虽然人们已提出大量聚类算法，如链接聚类或 DIANA，但最常用的聚类算法之一仍是 *K* 均值聚类。利用一个预定义的 *K* 值，即希望将数据划分为簇的数目，*K* 均值聚类算法可找到各簇的最优质心。*K* 均值聚类的最佳特质是各簇在本质上呈紧致的球状分布，且总会收敛到某个解。

接下来简要介绍 *K* 均值聚类的工作原理。

8.2.1　*K*均值算法

K 均值算法首先从某个基本配置开始。该算法随机从数据集中选择 *K* 个数据点，并将其定义为各簇的质心。接着，将数据集中的每个点分配到与每个不同的质心距离最近的那个簇中。这样，我们便得到了一个由初始随机质心确定的聚类结果。但这显然不是我们想要的结果，因此我们重新计算各簇的均值来更新质心。然后，我们重复上述过程，直到质心的位置不再发生变动为止（参见图 8-1）。

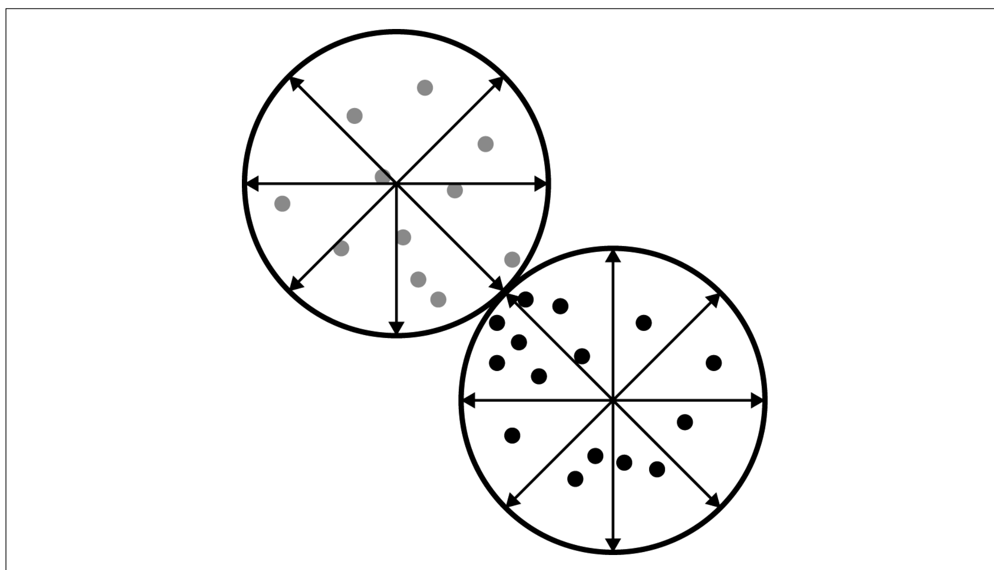


图 8-1：*K* 均值聚类的结果中各簇呈球状分布

K 均值中的距离可用不同的方式（第 3 章曾介绍过）来计算：

- 曼哈顿距离

$$d_{\text{manhattan}}(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

- 欧氏距离

$$d_{\text{euclid}}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- Minkowski距离

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

- Mahalanobis距离

$$d(x, y) = \sqrt{\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{s_i^2}}$$

8.2.2　*K*均值聚类的缺陷

K 均值聚类算法的缺陷是各簇之间必须存在一个“硬”边界。这意味着每个数据点只能属于一个簇，无法跨越两个簇之间的界限。此外，*K* 均值算法适合于呈球状分布的数据，因为大多数情况下人们采用的都是欧氏距离。在像图 8-1 这样的数据分布图中（位于中间的

那些点实际上可以属于两个簇的任意一个），这些缺陷非常明显。

8.3 EM聚类算法

EM 聚类算法的重点不是如何找到一个质心，然后找到与其相关的数据点，而是求解另一个不同的问题。比如你希望将一个数据集分为两部分：簇 1 和簇 2。你希望得到一个关于数据是否在某个簇中的良好估计，但并不关心其中是否存在模糊性。我们真正希望获得的是一个数据点属于各簇的概率值，而非分配结果。

与专注于确定各簇之间边界的 K 均值聚类算法不同，EM 聚类对于可能同属于多个簇的数据点具有一定的稳健性。EM 聚类算法非常适用于对不存在明确边界的数据进行分类。

在开始执行 EM 聚类时，我们先构造一个向量，z_k=<0.5, 0.5>，它保存的是行向量 k 属于各簇的概率。经过若干轮迭代，我们发现结果变为 z_k=<0.9, 0.1>。设想你有大量数据点，我们并不像之前那样用圆圈，而是用阴影来标识各簇。颜色越深，表示属于第 2 簇的可能性越大，而浅灰色表示第 1 簇，如图 8-2 所示。

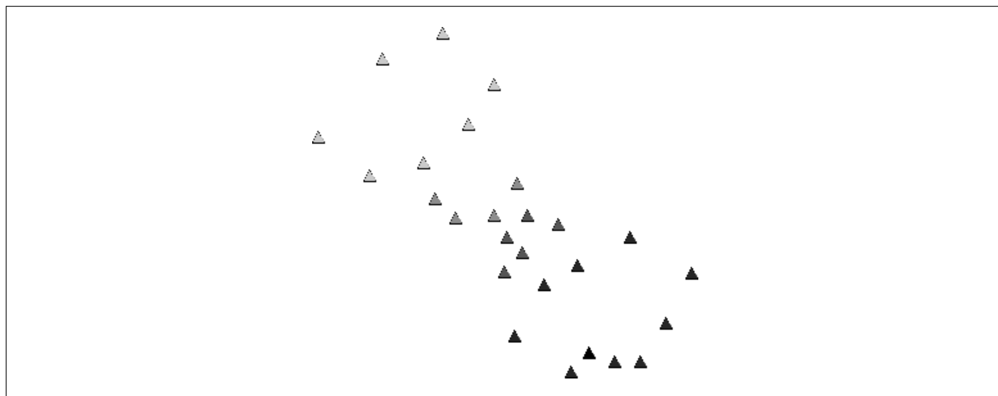


图 8-2：EM 聚类表明各簇之间的界限可以不必那样明确

EM 算法涉及两个步骤：计算期望和将期望最大化。

计算期望是给定数据的分布和初始 z_k，计算概率值。我们依据对参数 θ 的当前估计，计算关于条件分布 $Z|X$ 的对数似然函数：

$$Q(\theta \| \theta_t) = E_{Z\|X,\theta_t} \log L(\theta;X,Z)$$

接下来，将期望最大化，即找到给定 θ 时能够使关于 θ 的概率最大化的参数 θ_t：

$$\theta_t = \arg\max_{\theta} Q(\theta \| \theta_t)$$

EM 聚类算法的不足之处在于它无法保证收敛性，且当你使用奇异的协方差映射数据时，它可能会来回震荡。在示例分析一节中，我们将更加深入地探讨与 EM 聚类算法相关的问题。不过，我们首先需要讨论所有聚类算法所共有的一个特征——*不可能性定理*。

8.4 不可能性定理

天下没有免费的午餐，聚类算法也不例外。我们虽然能够借助聚类算法将数据点映射到某些簇中，但这是有代价的。这种代价可由 Jon Kleinberg 所提出的不可能性定理来概括。

该定理指出，我们最多只能获得下列属性中的两种：

- 丰富性
- 尺度不变性
- 一致性

*丰富性*是指存在某个距离函数，它可生成任意类型的划分。直观上看，这表示一个聚类算法能够创建任意类型的从数据点到簇的映射。

*尺度不变性*非常好理解。设想你正在构建一艘火箭飞船，在计算中你使用的单位是千米，但你的老板要求你将单位改为英里。这种单位转换不存在任何问题；你只需将所有的相关量除以某个常数即可。从而具有了尺度不变性。大体上讲，如果所有的数字都乘以 20，则聚类得到的簇不会有任何变化。

*一致性*略为微妙。与尺度不变性类似，如果我们将同一簇内的点对距离减小或增大，都不会影响最终的聚类结果。至此，你可能已经理解了聚类并不像你最初设想的那样出色。这类算法存在很多问题，一致性绝对是应当引起重视的问题之一。

K 均值聚类和 EM 聚类满足丰富性和尺度不变性，但不具有一致性。这使得聚类测试几乎不可能实现。我们真正能够进行测试的唯一途径是借助经验和示例。但对于分析而言这已经足够了。

在下一节中，我们将来利用 *K* 均值聚类和 EM 聚类来分析爵士乐。

8.5 音乐归类

音乐和作曲具有悠久的历史。你可能需要获得相关的专业学位，并对音乐理论进行研究，才能对这些乐谱进行有效的归类。

对音乐归类的方式不计其数。就本人而言，我通常会依据艺术家的名字对唱片进行归类。但不同的艺术家可能经常会合作。其他人则倾向于依据风格进行归类。但音乐风格非常广泛，要对音乐进行有效归类绝非易事。以爵士乐为例，从蒙特勒爵士音乐节（http://

www.montreuxjazzfestival.com/en/about-montreux-jazz）来看，很难对爵士乐的风格做一具体的限定。

那么我们如何有效地构建一个依据作品相似性自动归类的音乐库？



安装说明

本例所使用的全部代码均可从 GitHub 下载：https://github.com/thoughtfulml/examples/tree/master/7-expectation-maximization-clustering。

在本节中，我们将首先确定数据来源、所要提取的属性以及验证的依据。我们还将讨论为什么聚类虽然理论上看起来非常完美，但实践中并不能为我们提供充分的信息（除各簇外）。

由于 Ruby 处于持续的变化之中，因此要想获悉如何快速上手这些例子，README 文件无疑是最佳选择。

下面我们分别利用 *K* 均值聚类和 EM 聚类来求解这个问题。最后将得到一个乐曲的软聚类结果，我们可利用它构建一个音乐的分类法。



编写聚类程序时，我们不打算使用测试驱动的方法，因为聚类这个问题不适合进行假设检验。这是非常关键的一点。表面上，聚类看上去可以为所有问题提供优秀的解决方案，但实际上，它并不适合对我们的假设进行检验。

通过之前对不可能性定理的讨论，我们知道要想获得一个同时满足一致性、丰富性和尺度不变性的聚类算法是不可能的（至多只能满足其中两点）。从许多方面来看，聚类方法只是一个数据分析工具，并不能用来解决我们希望可控的问题。

8.5.1 数据收集

从 12 世纪至今，人类已经积累了规模极其可观的音乐数据。MP3、CD、黑胶唱片以及手写的乐谱，种类繁多。我们并不打算对所有的音乐数据进行分类，而是从中挑选一个规模很小的子集。我不希望卷入任何版权纠纷，因此我们将仅使用唱片集中的公开信息，包括艺术家、歌曲名、风格（如果有的话）以及我们能够在音乐中找到的其他特征。为此，我们将访问 Discogs.com 所收集的信息，其中包括大量录音和歌曲的 XML 数据转储。

另外，我们并不打算对已有的全部唱片集进行聚类，而是只关注爵士乐。大多数人都认同爵士乐是一种难以归到任何类别的风格。爵士乐既有可能是混合音乐，也有可能是用钢鼓演奏，种类繁多。

因此，为了获取数据集，我依据 http://www.scaruffi.com/jazz/best100.html 网站下载了最受欢迎的爵士乐唱片集。

这些数据的发行时间范围为 20 世纪 40 年代至 21 世纪初。我一共下载了约 1200 张不同的唱片。这是一个规模多么庞大的唱片集！

但其中的信息尚不充分。此外，我利用 Discogs API 对这些唱片进行了信息标注，以确定每张爵士乐唱片的风格。

对原始数据集标注完成之后，我发现与爵士乐有关的音乐风格共有 128 种（至少是依据 Discogs），从土著音乐到噪音，其范围极为广阔。

8.5.2　用*K*均值聚类分析数据

如同使用 *K* 近邻算法一样，我们也需要预先确定一个最优的 *K* 值。不幸的是，对于聚类，我们所能做的测试不多，只能简单地查看是否可将数据集分为两个不同的簇。

假设我们希望将所有的唱片上架，而书架上共有 25 个格子。我们可将 *K* 取为 25，然后对数据集进行聚类。

要完成聚类，只需写极少量的代码，因为我们可利用 ai4r gem 所提供的功能：

```
# lib/kmeans_clusterer.rb

require 'csv'
require 'ai4r'

data = []

artists = []
CSV.foreach('./annotated_jazz_albums.csv', :headers => true) do |row|
  @headers ||= row.headers[2..-1]
  artists << row['artist_album']
  data << row.to_h.values[2..-1].map(&:to_i)
end

ds = Ai4r::Data::DataSet.new(:data_items => data, :data_labels => @headers)
clusterer = Ai4r::Clusterers::KMeans.new
clusterer.build(ds, 25)

CSV.open('./clustered_kmeans.csv', 'wb') do |csv|
  csv << %w[artist_album year cluster]
  ds.data_items.each_with_index do |dd, i|
    csv << [artists[i], dd.first, clusterer.eval(dd)]
  end
end
```

没错，就是这么简单！当然，如果不观察输出结果，聚类算法便毫无意义。这段代码的确

可以将数据划分到 25 个类别中，但这些类别反映了什么样的信息呢？

图 8-3 制了年份与簇的关系，呈现出非常有趣的结果。

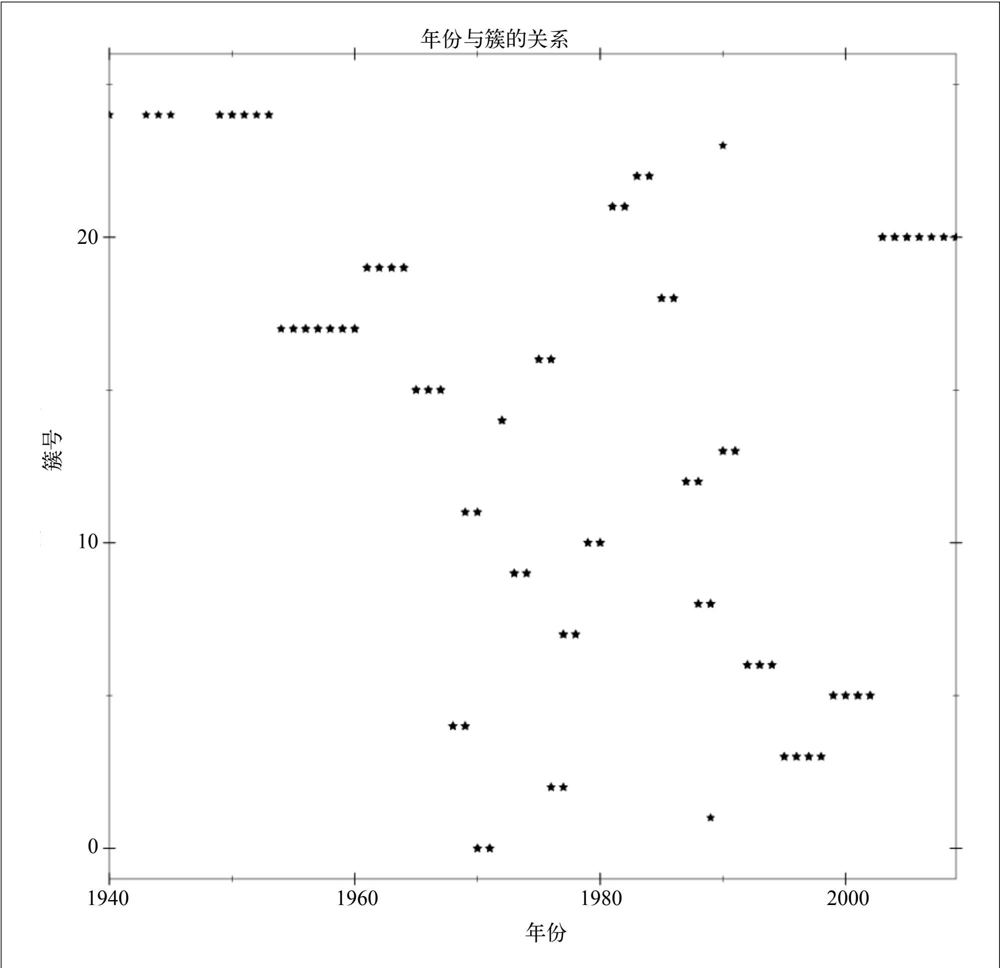


图 8-3：将 *K* 均值运用于爵士乐唱片集

如你所见，爵士乐始于大乐队时代，这个时期的爵士乐几乎都位于同一个簇中。此后，它演变为冷爵士乐。接着，到了大约 1959 年，它开始朝着许多不同的方向发展，一直到 20 世纪 90 年代，冷爵士乐开始降温。

我们的聚类结果居然与爵士乐的历史保持了惊人的同步，这是多么奇妙的一件事！

如果我们利用 EM 聚类算法，会得到什么样的结果？

8.5.3 EM聚类

使用 EM 聚类算法时，请牢记我们依某个概率对数据所属的簇进行分配：不存在 100% 属于某个簇的数据。对于我们而言，这是极有价值的，因为有太多的爵士乐跨界了。

由于 Ruby gem 中没有相应的 EM 聚类实现，因此下面我们来编写一个自己的版本。

下面我们来一步步构建自己的 gem，并利用它对从爵士乐数据集中选取的同一批数据进行映射。

第一步是初始化各簇。请牢记，你需要维护一组服从均匀分布的指示变量 Z*t*。这些变量所记录的是每个数据点属于各个簇的概率。为此，我们有：

```
# lib/em_clusterer.rb

require 'matrix'

class EMClusterer
  attr_reader :partitions, :data, :labels, :classes

  def initialize(k, data)
    @k = k
    @data = data
    setup_cluster!
  end

  def setup_cluster!
    @labels = Array.new(@data.row_size) { Array.new(@k) { 1.0 / @k }}

    @width = @data.column_size
    @s = 0.2

    pick_k_random_indices = @data.row_size.times.to_a.shuffle.sample(@k)

    @classes = @k.times.map do |cc|
      {
        :means => @data.row(pick_k_random_indices.shift),
        :covariance => @s * Matrix.identity(@width)
      }
    end
    @partitions = []
  end
end
```

至此，我们已经完成了基础代码的编写。我们有 `@k`，簇的数目；`@data`，我们希望对其进行聚类的数据；`@labels`，一个由概率构成的数组，每行表示当前行所代表的数据属于各个簇的概率；`@classes`，记录了各簇的均值和方差，表明了各簇的数据分布情况。此外，还有 `@partitions`，每一行都标识了一个数据点最终的簇归属。

现在我们需要计算期望，即确定每个数据行对各簇的概率。为此，我们需要编写一个新方法 `expect`：

```ruby
# lib/em_clusterer.rb

class EMClusterer
  # initialize
  # setup_cluster!

  def expect
    @classes.each_with_index do |klass, i|
      puts "Expectation for class #{i}"

      inv_cov = if klass[:covariance].regular?
        klass[:covariance].inv
      else
        puts "Applying shrinkage"
        (klass[:covariance] - (0.0001 * Matrix.identity(@width))).inv
      end

      d = Math::sqrt(klass[:covariance].det)

      @data.row_vectors.each_with_index do |row, j|
        rel = row - klass[:means]

        p = d * Math::exp(-0.5 * fast_product(rel, inv_cov))
        @labels[j][i] = p
      end
    end

    @labels = @labels.map.each_with_index do |probabilities, i|
      sum = probabilities.inject(&:+)

      @partitions[i] = probabilities.index(probabilities.max)

      if sum.zero?
        probabilities.map { 1.0 / @k }
      else
        probabilities.map {|p| p / sum.to_f }
      end
    end
  end

  def fast_product(rel, inv_cov)
    sum = 0

    inv_cov.column_count.times do |j|
      local_sum = 0
      (0 ... rel.size).each do |k|
        local_sum += rel[k] * inv_cov[k, j]
      end
      sum += local_sum * rel[j]
    end

    sum
  end
end
```

第一部分遍历了所有的类别，并记录了各簇的均值和方差。从这里，我们希望找到逆协方差矩阵以及协方差矩阵的行列式。对于每一行，我们都计算出一个与当前行所对应数据位于某个簇的概率成比例的值：

$$p_{ij} = \det(C) e^{-\frac{1}{2}(x_j - \mu_i)C^{-1}(x_j - \mu_i)}$$

这实际上是一个高斯距离度量，可帮助我们确定我们的数据偏离均值的情况。

假设行向量等于均值，则意味着该值简化为$p_{ij} = \det(C)$，即协方差矩阵的行列式。这实际上是该函数能够取得的最大值。如果行向量远离均值向量，则由于指数为负，p_{ij}将变得很小很小。

好处是这个值与行向量到均值的高斯概率成正比。由于这里只是成正比，而非相等，因此需要归一化，以使其和为 1。

你可能也注意到了最后一点：`fast_product` 方法的引入。这是因为 Ruby 的矩阵运算效率很低，且矩阵是使用 Array 嵌套构成的，内存效率低下。在本例中，考虑到数据维数和规模不变，我们可对其进行优化。

现在，我们来实现期望最大化：

```ruby
# lib/em_clusterer.rb

class EMClusterer
  # initialize
  # setup_cluster!
  # expect
  # fast_product

  def maximize
    @classes.each_with_index do |klass, i|
      puts "Maximizing for class #{i}"
      sum = Array.new(@width) { 0 }
      num = 0

      @data.each_with_index do |row, j|
        p = @labels[j][i]

        @width.times do |k|
          sum[k] += p * @data[j,k]
        end

        num += p
      end

      mean = sum.map {|s| s / num }
      covariance = Matrix.zero(@width, @width)

      @data.row_vectors.each_with_index do |row, j|
```

```ruby
        p = @labels[j][i]
        rel = row - Vector[*mean]
        covariance += Matrix.build(@width, @width) do |m,n|
          rel[m] * rel[n] * p
        end
      end

      covariance = (1.0 / num) * covariance

      @classes[i][:means] = Vector[*mean]
      @classes[i][:covariance] = covariance
    end
  end
end
```

我们再一次对各簇 `@classes` 进行遍历。首先构造一个名为 `sum` 的数组，以保存数据的加权和。从这时起，我们通过归一化来构建一个加权均值。为计算协方差矩阵，我们首先从一个行数和列数均为簇的宽度的零方阵开始，然后对所有 `row_vectors` 进行迭代，并为该矩阵的每个组合增量式地添加行向量和均值的加权差。同样，此后需要归一化并保存。

现在我们来实际使用一下这些代码。为此，我们可添加两个辅助方法来帮助查询数据：

```ruby
# lib/em_clusterer.rb

class EMClusterer
  # initialize
  # setup_cluster!
  # expect
  # fast_product
  # maximize

  def cluster(iterations = 5)
    iterations.times do |i|
      puts "Iteration #{i}"
      expect_maximize
    end
  end

  def expect_maximize
    expect
    maximize
  end
end
```

8.5.4　爵士乐的EM聚类结果

我们再回到用 EM 聚类算法对爵士乐数据聚类的结果。为进行分析，可运行下列脚本：

```ruby
data = []

artists = []
```

```
CSV.foreach('./annotated_jazz_albums.csv', :headers => true) do |row|
  @headers ||= row.headers[2..-1]
  artists << row['artist_album']
  data << row.to_h.values[2..-1].map(&:to_i)
end

data = Matrix[*data]


e = EMClusterer.new(25, data)
e.cluster
```

关于 EM 聚类算法，我们首先注意到的是它的运行效率很低。它不像计算新的质心和迭代那样高效。它必须计算协方差和均值，这种计算开销是很大的。奥卡姆剃刀准则告诉我们，在对大量数据分组时，使用 EM 聚类算法并不是一个很好的选择。

你还会注意到，EM 算法对我们的带标注的爵士乐数据是不适用的。这是因为协方差是奇异的。这并不是一件好事。实际中，出于这个原因，这个问题并不适合使用 EM 算法来求解，因此我们需要将其变换为一个完全不同的问题。

我们可对维数进行压缩，只保留按索引序的前两种风格：

```
require 'csv'

CSV.open('./less_covariance_jazz_albums.csv', 'wb') do |csv|
  csv << %w[artist_album key_index year].concat(2.times.map {|a| "Genre_#{a}" })

  CSV.foreach('./annotated_jazz_albums.csv', :headers => true) do |row|
    genre_count = 0
    genres = Array.new(2) { 0 }
    genre_idx = 0

    row.to_h.values[4..-1].each_with_index do |g, i|
      break if genre_idx == 2
      if g == '1'
        genres[genre_idx] = i
        genre_idx += 1
      end
    end

    next if genres.count {|a| a == 0 } == genres.length

    csv << [row['artist_album'], row['key_index'], row['year']].concat(genres)
  end
end
```

这里，我们基本上是在讲，对于前两个 Genres，我们为其分配一个风格索引，并保存。接下来的问题是，有些唱片集没有可以分配的信息，因此我们将其忽略。

至此，我们已经能够运行 EM 聚类算法，只是要想形成一些簇非常困难。这是使用 EM

聚类算法的一个深刻教训。我们的数据之所以无法形成簇，是因为协方差矩阵稳定性太差，而无法求逆。我将此作为练习留给你，你可观察程序何时失效，协方差矩阵何时变得不可逆。

8.6 小结

聚类非常有用，但由于它是无监督的，因此可控性较差。考虑到一致性、丰富性和尺度不变性无法求全的不可能性定理，聚类在许多场景中又显得没有价值。但请不要因此便对聚类丧失信心：聚类对于数据集分析和将数据划分为任意数量的簇非常有用。如果你并不关心数据划分的过程，而只是希望将数据分组，则聚类便可派上用场。只是要注意，有时会出现一些非常奇怪的情形。

第9章

核岭回归

回归可能是任意机器学习工具集中最为常见的工具。回归就是从将 X 映射为 Y 的数据中拟合出一条直线。你可能已经接触过大量的回归问题。在许多方面，回归都是对最常见的情形和基本情形进行建模。通过本章的学习，你将了解到，线性回归是预测数据的一个良好起点，但当你试图对小规模或非线性的数据集建模时，线性回归将会迅速失效。

我们首先介绍协同过滤问题和推荐算法，然后待介绍岭回归时再详细介绍如何求解该问题。最后，在本章的结尾，我们将通过示例确定我们的假设是否成立。



回归，包括核岭回归算法，是一种有监督学习算法。它对于能够求解的问题限制很少，但更擅长使用连续变量。这种算法还有一个好处是能够对数据进行平滑并移除离群点。

9.1 协同过滤

如果你从 Amazon 网站购买过商品，说明你已经接触过协同过滤算法了。对于 Amazon，它希望向你推荐你感兴趣的商品，以使你购买更多的商品。因此，假设你购买了许多啤酒，则向你推荐一些啤酒是比较好的选择。

但协同过滤的真正有趣之处，是它与其他用户关联的方式。假设你喜欢喝啤酒，那你可能也喜欢小桶、杯子甚至是杯垫。即便你没有实际购买过这些物品，但由于与你类似的其他用户（也喜爱喝啤酒）购买了它们，因此你也很可能喜欢它们。

协同过滤的图形化表示如图 9-1 所示。

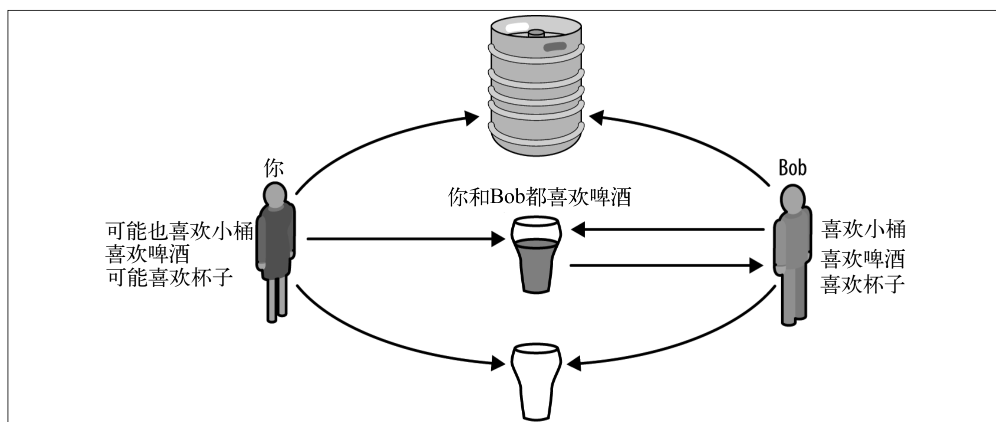


图 9-1：你和 Bob 都喜欢啤酒，因此你可能也喜欢小桶和杯子

任何协同过滤算法通常都由两部分构成：

- 找到那些与你有相同爱好的用户（例如其他爱喝啤酒的人）；
- 利用来自其他相似用户的评分来进行推荐。

我们通常可通过如下三种方式来解决这些问题：

- 暴力法，这种朴素方法的时间复杂度与数据规模呈指数增长；
- *K* 近邻搜索，第 2 章已介绍过；
- 回归。

暴力法是一种基准方法。你可遍历所有可能的连接，以对用户生成最优推荐。但考虑到用户很可能希望每个页面中都有大量推荐商品，这种策略的效率势必无法满足要求。*K* 近邻搜索算法效果不错。由于它是一种“懒惰”的方法，因此它没有前期的计算开销，但与此同时，你实际上是对数据做出了某种假设，而且通过这种算法除了能了解到当前用户与其他 *K* 个用户相似外，并不会产生更多信息。

最后，回归方法可生成一条拟合了特征到用户是否喜欢某种商品的直线。这种方法的优点在于，我们可依据回归系数来确定用户喜欢什么，并利用矩阵代数快速求解。这可能是最好的方案，而且非常简单。

9.2　应用于协同过滤的线性回归

你可能对线性回归早有耳闻。其思想非常简单：给定一个数据集，拟合出逼近这些数据的最优直线。例如，我们希望依据身高预测体重。从概念上，我们了解身高和体重至少是相

关的（参见图 9-2）。下面以参加 2012 年伦敦奥运会的 11 000 名运动员的数据为例。

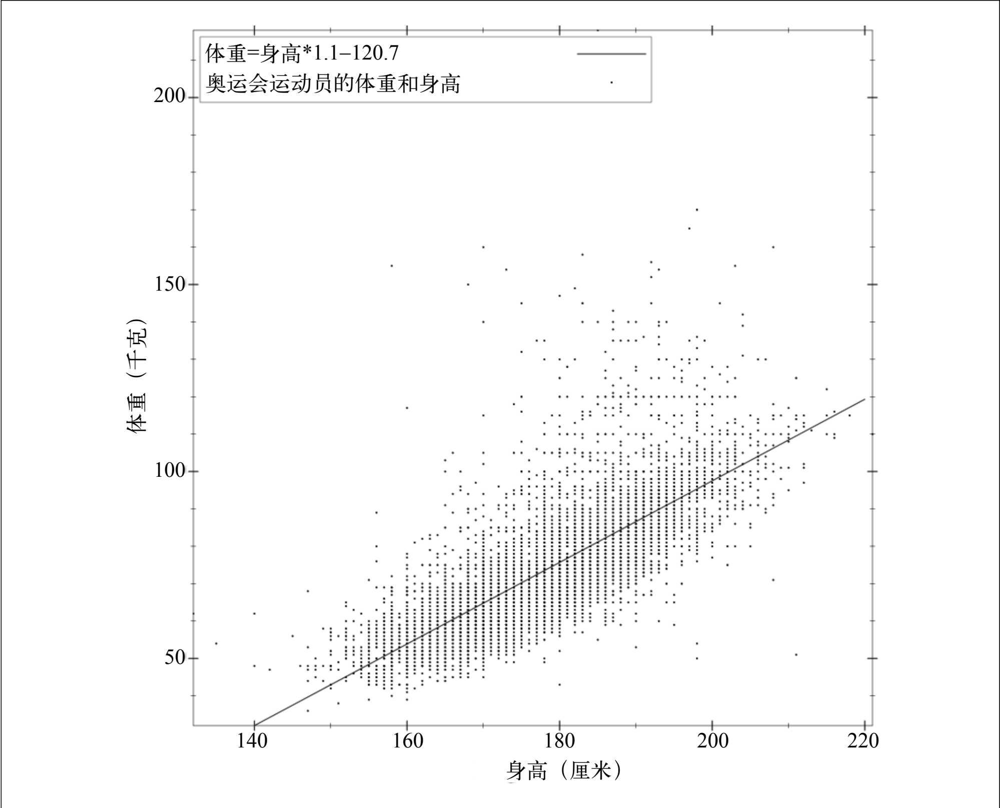


图 9-2：体重可视为身高的函数（可拟合出一条服从数据分布的直线）

这是有意义的：如果我的身高为 5.6 寸（1.68 米），则体重很可能不到 200 磅（91 千克）。另一方面，如果我的身高为 6.8 寸（2.03 米），则体重可能超过 250 磅（113.4 千克）。当然，这需要将大量离群点移除，但这是完全可行的。回归的要旨在于*回归到均值*；最终得到的那条直线实际上代表了大多数情况下得到的数据点的均值。下面我们来了解线性回归的一些技术细节，它的主要目标是将数据的均方误差最小化，即：

$$\min \sum_{i}^{n} (\hat{y} - y)^2$$

其中，$\hat{y}$为回归模型的输出。对于线性回归，$\hat{y}$的形式为$\hat{y} = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$。这些$\beta$是我们求出的能够将上述均方误差最小化的系数。

线性回归模型的真正威力在于，为找到这些系数β，我们只需对原始矩阵进行一个简单的变换。

利用 Moore-Penrose 伪逆，我们可利用下式求得上述优化问题的解：

$$\vec{\beta} = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y$$

从根本上说，我们只需对训练数据 X 做转置运算（即将矩阵的第 i, j 个元素与第 j, i 个元素交换），并乘以自身 X。该乘积总是一个方阵，我们可利用矩阵求逆来得到其逆矩阵（由于超出了本书的范围，我不打算介绍其中的实现细节）。最后，我们将这个逆矩阵再乘以训练数据的转置。最终得到一个能够最优地匹配训练数据的系数向量。

该公式中最精彩之处在于，你可有效地得到一个给定数据的平均解。假设现有一个一维问题，数据点为 1、2、3、4、5、6，我们的目标是找到一个最优点。我们只需令所有点的 x 坐标为 1，y 坐标为 1~6，便可将这个问题映射为一个二维问题。对于该问题，找到的 β 值会是怎样的？

```
require 'matrix'

y = Matrix[[1],[2],[3],[4],[5],[6]]
x = Matrix[[1],[1],[1],[1],[1],[1]]

(x.transpose * x).inverse * x.transpose * y #=> Matrix[[7/2]]
```

这样便可得到均值！这里再一次说明，线性回归所做的不过是回归到均值。

与线性回归有关的一个重要问题是矩阵与其自身转置之积有时是奇异的（即不可逆）。奇异矩阵也称为病态问题（`ill_conditioned`），这是因为对于任意方程都缺少足够的数据来得到一个合理的解。这类矩阵的行列式为 0。

下面给出一个用 Ruby 代码表示奇异矩阵的例子：

```
require 'matrix'

y = Matrix[[1],[2]]

ill_conditioned = Matrix[[1,2,3,4,5,6,7,8,9,10], [10,9,8,7,6,5,4,3,2,1]]
(ill_conditioned.transpose * ill_conditioned).singular? #=> true
(ill_conditioned.transpose * ill_conditioned).inverse #=> Throws an error

conditioned = Matrix[[1,2], [2,1]]
(conditioned.transpose * conditioned).inverse * conditioned.transpose * y #=> Ma
trix[[(1/1)], [(0/1)]]
```

这里所展示的是，如果你像第一个病态问题一样拥有大量变量，则利用线性回归将没有合适的方法来求解该问题，因为数据点的数量太少，不足以找到最优的最小二乘解。当特征的数量多于数据点时，内部矩阵将成为奇异阵。

关于这个问题，我们不妨做一点更深入的思考。

9.3 正则化技术与岭回归

上述病态回归问题可用表 9-1 概括。

表9-1：病态问题

Y	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
1	1	2	3	4	5	6	7	8	9	10
2	10	9	8	7	6	5	4	3	2	1

即便没有任何算法来求解这个问题，我们也能从中看出有大量数据是无用的。我们所要寻找的是一个可在第一种情形中得到 1、在第二种情形中得到 2 的函数，因此我们可能希望找到 2 倍于第二种情形而 1 倍于第一种情形的信息。这意味着像 X_1、X_2、X_5、X_6、X_9 和 X_{10} 这样的列都是无用的。因此我们将之移除，如表 9-2 所示。

表9-2：列数较少的病态问题

Y	X_3	X_4	X_7	X_8
1	3	4	7	8
2	8	7	4	3

实际上，X_7 和 X_8 的用处也不大，因此我们将它们一并移除。这样便只剩下 X_3 和 X_4。现在，我们可这样来求解：

```
require 'matrix'

y = Matrix[[1],[2]]
simplified = Matrix[[3,4], [8,7]]

betas = (simplified.transpose * simplified).inverse * simplified.transpose * y
# => Matrix[[(1/11)], [(2/11)]]
```

这个问题就这样迎刃而解！我们所做的仅是将一些没有关系的变量移除。但问题在于，我们能否通过某种算法而非人工方式来自动求解？

是的，我们的确可以——我们可以利用一种称为*核岭回归*（或*正则化回归*）的算法。

该算法的基本思想是引入一个岭参数，它将有助于解决我们之前遇到的病态问题：

```
require 'matrix'
y = Matrix[[1],[2]]
ill_conditioned = Matrix[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]

shrinkage = 0.0001

left_half = ill_conditioned.transpose * ill_conditioned + shrinkage * Matrix.ide
ntity(ill_conditioned.column_size)
```

```
left_half.singular? #=> false

betas = left_half.inverse * ill_conditioned.transpose * y

betas.transpose * ill_conditioned.row(0) #=> Vector[1.0000000549097194]
betas.transpose * ill_conditioned.row(1) #=> Vector[1.9999994492261521]
```

如你所见，增加收缩因子有助于我们克服奇异矩阵所引发的问题，而且可轻松地求解这个病态问题。但此时仍有一个问题有待解决，即非线性。

9.4 核岭回归

在第 6 章中我们曾介绍过核技巧，它可将非线性数据变换到一个新的特征空间中，使其成为线性数据。核技巧是非常强大的工具，非常适用于求解数据非线性的问题。

下面我们简要回顾一下第 6 章介绍过的几个核函数：

- 齐次多项式

$$K(x_i, x_j) = (x_i^{\mathrm{T}} x_j)^d$$

- 非齐次多项式

$$K(x_i, x_j) = (x_i^{\mathrm{T}} x_j + c)^d$$

- 径向基函数

$$K(x_i, x_j) = \mathrm{e}^{-\frac{\|x_i - x_j\|^2}{2*\sigma^2}}$$

这里需要了解的很重要的一点是，实际上我们不需要进行大量计算，因为这些函数的主要运算来自 $X^{\mathrm{T}}X$，而这个量是可以事先计算好的。大体上说，如果不去细究数学细节，我们可将 $X^{\mathrm{T}}X$ 记为 K，它是一个代表了新的非线性空间的核。

这样，我们的方程看起来都很相似：

$$\hat{y} = y^{\mathrm{T}}(K + \lambda I)^{-1} k$$

$$K_{ij} = f(x_i, x_j)$$

$$k_i = f(x_i, x')$$

这只是一点改变。你可将这些函数添加到之前的函数中，从而使线性回归模型拥有了核函数，这与第 6 章的用法很像。

9.5 理论总结

核岭回归算法可用于找到一个简单函数来映射一个病态问题。在下一节中，我们将了解如

何实际使用该算法，依据用户在评价中所表达的偏好向他们推荐不同风味的啤酒。

9.6 用协同过滤推荐啤酒风格

你是否还记得本章开头提到的用协同过滤向爱喝啤酒的客户推荐商品？如果将协同过滤运用于关于啤酒风格的真实数据集，应如何实现？所有的评论者都会在商品评价中提到他们是否喜欢这种口味、外观及其他属性。

9.6.1 数据集

我们所使用的数据集中包含了啤酒风格、啤酒、啤酒厂、评论者及评论。共有 1 586 615 条评价，62 260 种不同的啤酒，33 388 名评论者、5734 家啤酒厂以及 104 种不同的啤酒风格。在此之前，我们一直是将所有数据都加载到内存中，然后进行分析，但由于这个数据集的规模较大，一种更好的方法是将这些信息加载到某种类型的数据库中。



安装说明

本例中使用的所有代码均可从 GitHub 获取：https://github.com/thoughtfulml/examples/tree/master/8-kernel-ridge-regression。

由于 Ruby 处于持续变化之中，因此要想获悉如何快速上手这些例子，README 文件无疑是最佳选择。

为何用回归来实现协同过滤？

如果你阅读过其他机器学习或数据科学方面的书籍，可能并不清楚回归还可以运用到协同过滤上。在大多数情况下，人们会运用一种称为**矩阵分解**的技术来进行推荐。

本例中我们之所以使用回归是因为它非常适用于确定各种因素的线性组合系数，这些因素确定了某人的购买需求。其好处是，我们可以用其确定某人的偏好。因此，在关于啤酒的评价中，我们可明确某人是喜欢酒精还是喜欢某种口味。虽然我们也可以利用矩阵分解来完成这一任务，但这两种方法还是存在一些差异的。

9.6.2 我们所需的工具

为了实现对啤酒风格的协同过滤，我们需要将一些表格存入 Postgres。我认为对于像我们这样的小型项目，Sequel 要比 ActiveRecord 易用得多，因此我们将使用前者。

首先，我们来定义一些将要使用的表格和模型。我们需要关于啤酒、评价、评论者、啤酒厂以及啤酒风格的表格。

我们先来创建文件 bootstrap，以确保这些表格存在，且所有数据都被正确地迁移：

```
# script/load_db.rb

# 注意,这里并非只能为Postgres,你也可以使用sqlite或mysql

DB = Sequel.connect('postgres://localhost/beer_reviews')

DB.create_table? :beers do
  primary_key :id
  Integer :beer_style_id, :index => true
  Integer :brewery_id, :index => true
  String :name
  Float :abv
end
```

啤酒拥有的属性包括 beer_style_id、name、abv 以及 brewery_id。我们希望这些信息能够向外扩展，因此将 beer_style_id 作为 beer_style 的外键，abv 表示酒精的体积，而 brewery_id 是啤酒厂的外键。接下来构建自己的啤酒厂以及其他信息：

```
# script/load_db.rb

# DB
# create_table :beers

DB.create_table? :breweries do
  primary_key :id
  String :name
end

DB.create_table? :reviewers do
  primary_key :id
  String :name
end

DB.create_table? :reviews do
  primary_key :id
  Integer :reviewer_id, :index => true
  Integer :beer_id, :index => true
  Float :overall
  Float :aroma
  Float :appearance
  Float :palate
  Float :taste
end

DB.create_table? :beer_styles do
  primary_key :id
  String :name, :index => true
end
```

现在我们需要加载这些信息，可通过下列脚本来完成：

```ruby
# script/load_db.rb

# DB
# create_table :beers
# create_table :breweries
# create_table :reviewers
# create_table :reviews
# create_table :beer_styles

require 'csv'
require 'set'

# brewery_id,brewery_name,review_time,review_overall,review_aroma,review_appeara
nce,review_profilename,beer_style,review_palate,review_tast,beer_name,beer_abv,b
eer_beerid
breweries = {}
reviewers = {}
beer_styles = {}

if !File.exists?('./beer_reviews/beer_reviews.csv')
  system('bzip2 -cd ./beer_reviews/beer_reviews.csv.bz2 > ./beer_reviews/beer_
reviews.csv') or die
end

CSV.foreach('./beer_reviews/beer_reviews.csv', :headers => true) do |line|
  puts line
  if !breweries.has_key?(line.fetch('brewery_name'))
    b = Brewery.create(:name => line.fetch('brewery_name'))
    breweries[line.fetch('brewery_name')] = b.id
  end

  if !reviewers.has_key?(line.fetch('review_profilename'))
    r = Reviewer.create(:name => line.fetch('review_profilename'))
    reviewers[line.fetch('review_profilename')] = r.id
  end

  if !beer_styles.has_key?(line.fetch('beer_style'))
    bs = BeerStyle.create(:name => line.fetch('beer_style'))
    beer_styles[line.fetch('beer_style')] = bs.id
  end

  beer = Beer.create({
    :beer_style_id => beer_styles.fetch(line.fetch('beer_style')),
    :name => line.fetch('beer_name'),
    :abv => line.fetch('beer_abv'),
    :brewery_id => breweries.fetch(line.fetch('brewery_name'))
  })

  Review.create({
    :reviewer_id => reviewers.fetch(line.fetch('review_profilename')),
    :beer_id => beer.id,
    :overall => line.fetch('review_overall'),
    :aroma => line.fetch('review_aroma'),
    :appearance => line.fetch('review_appearance'),
    :palate => line.fetch('review_palate'),
```

```
    :taste => line.fetch('review_taste')
  })
end
```

既然我们已经加载了数据，接下来便可利用岭回归来测试和构建推荐算法。

9.6.3　评论者

我们要做的第一步是快速建立所有模型之间的关联，为此可编写下列代码：

```
# lib/models/reviewer.rb

class Reviewer < Sequel::Model
  one_to_many :reviews
  one_to_many :user_preferences
end

# lib/models/brewery.rb
class Brewery < Sequel::Model
  one_to_many :beers
end

# lib/models/beer_style.rb
class BeerStyle < Sequel::Model
  one_to_many :beers
end

# lib/models/review.rb
class Review < Sequel::Model
  many_to_one :reviewer
end

# lib/models/user_preference.rb
class UserPreference < Sequel::Model
  many_to_one :reviewer
  many_to_one :beer_style
end
```

至此，我们需要为两个不同的场景编写测试。第一个场景是对于被评价的每个风格，我们希望为其赋予一个非零常量。第二个场景是我们希望最大的斜率对应最受欢迎的风格，并且最小的斜率对应受欢迎程度最低的啤酒风格。

为测试计算的正确性，可编写下列代码：

```
# test/lib/models/reviewer_spec.rb
describe Reviewer do
  let(:reviewer) { Reviewer.find(:id => 3) }

  it 'calculates a preference for a user correctly' do
    pref = reviewer.preference

    reviewed_styles = reviewer.reviews.map {|r| r.beer.beer_style_id }
```

```
    pref.each_with_index do |r,i|
      if reviewed_styles.include?(i + 1)
        r.wont_equal 0
      else
        r.must_equal 0
      end
    end
  end
end
```

我们只假设你将 Reviewer 加载到数据库中，且你可随机选择一个评论者进行测试，如 id=3。该测试将确保对于未被评价的风格，只有 0，而对于已评价的风格，则为非零常量。

至此，我们可对实际问题进行测试，即是否可对啤酒风格的受欢迎程度进行排序。我们通过编写下列代码来完成该任务：

```
# test/lib/models/reviewer_spec.rb

describe Reviewer do
  let (:reviewer) { Reviewer.find(:id => 3) }

  # 测试

  it 'gives the highest rated beer_style the highest constant' do
    pref = reviewer.preference

    most_liked = pref.index(pref.max) + 1

    least_liked = pref.index(pref.select(&:nonzero?).min) + 1

    reviews = {}
    reviewer.reviews.each do |r|
      reviews[r.beer.beer_style_id] ||= []
      reviews[r.beer.beer_style_id] <<  r.overall
    end

    review_ratings = Hash[reviews.map {|k,v| [k, v.inject(&:+) / v.length.to_
f] }]

    assert review_ratings.fetch(most_liked) > review_ratings.fetch(least_liked)

    best_fit = review_ratings.max_by(&:last)
    worst_fit = review_ratings.min_by(&:last)

    assert best_fit.first == most_liked || best_fit.last == review_ratings[most_
liked]
    assert worst_fit.first == least_liked || worst_fit.last == review_ratings[le
ast_liked]
  end
end
```

现在，为使上述代码能够正常工作，我们当然需要编写实际的代码。

9.6.4 编写代码确定某人的偏好

我们要利用这两项测试解决的问题是，找到啤酒风格的一个线性组合，以便对所有评分进行平均。总共约有 104 种啤酒风格，而大多数用户都不会去频繁地评价。因此，我们可能会得到一个奇异阵，而回归方法可能无法使用。因此，我们需要构建一个能充分压缩这个矩阵以使其可逆的算法。我们通过指数规律增长收缩系数，直到矩阵可逆为止。

你可能已经注意到，我使用了 NMatrix 库，它是 NArray 的一个子集。这纯粹是出于效率的考虑。不幸的是，Ruby 中矩阵库的效率非常低。因此，当计算量较大时，我们需要利用 NMatrix 库。当然，这个库也有一些缺点，即它只是对 NArray 进行了简单的包装，并不具备像行列式这样的特性或其他简洁的工具。因此，我创建了一个名为 `MatrixDeterminance` 的类，专门负责计算矩阵的行列式：

```ruby
# lib/matrix_determinance.rb

require 'narray'
require 'nmatrix'

class MatrixDeterminance
  def initialize(matrix)
    @matrix = matrix
  end

  def determinant
    raise "Must be square" unless square?
    size = @matrix.sizes[1]
    last = size - 1
    a = @matrix.to_a
    no_pivot = Proc.new{ return 0 }
    sign = +1
    pivot = 1.0
    size.times do |k|
      previous_pivot = pivot
      if (pivot = a[k][k].to_f).zero?
        switch = (k+1 ... size).find(no_pivot) {|row|
          a[row][k] != 0
        }
        a[switch], a[k] = a[k], a[switch]
        pivot = a[k][k]
        sign = -sign
      end
      (k+1).upto(last) do |i|
        ai = a[i]
        (k+1).upto(last) do |j|
          ai[j] = (pivot * ai[j] - ai[k] * a[k][j]) / previous_pivot
        end
      end
    end
    sign * pivot
  end
```

```
  def singular?
    determinant == 0
  end

  def square?
    @matrix.sizes[0] == @matrix.sizes[1]
  end

  def regular?
    !singular?
  end
end
```

该类的用法十分简单，我们只需初始化一个新的 `MatrixDeterminance` 对象，便可计算矩阵是否奇异，以及该矩阵的行列式。

现在，我们利用岭回归来计算用户的偏好：

```
# lib/models/reviewer.rb

class Reviewer < Sequel::Model
  one_to_many :reviews
  one_to_many :user_preferences

  IDENTITY = NMatrix[
    *Array.new(104) { |i|
      Array.new(104) { |j|
        (i == j) ? 1.0 : 0.0
      }
    }
  ]

  def preference
    @max_beer_id = BeerStyle.count
    return [] if reviews.empty?
    rows = []
    overall = []

    context = DB.fetch(<<-SQL)
      SELECT
        AVG(reviews.overall) AS overall
        , beers.beer_style_id AS beer_style_id
      FROM reviews
      JOIN beers ON beers.id = reviews.beer_id
      WHERE reviewer_id = #{self.id}
      GROUP BY beer_style_id;
    SQL

    context.each do |review|
      overall << review.fetch(:overall)
      beers = Array.new(@max_beer_id) { 0 }
      beers[review.fetch(:beer_style_id) - 1] = 1
```

```
      rows << beers
    end

    x = NMatrix[*rows]
    shrinkage = 0

    left = nil
    iteration = 6

    xtx = (x.transpose * x).to_f

    left = xtx + shrinkage * IDENTITY

    until MatrixDeterminance.new(left).regular?
      puts "Shrinking iteration #{iteration}"
      shrinkage = (2 ** iteration) * 10e-6

      (left * x.transpose * NMatrix[overall].transpose).to_a.flatten
    end
  end
end
```

可以看出，这个方法非常臃肿，但我们还是一起梳理一下。第一步是找到最大的 beer_style_id。最终向量的长度将是这个值。接着，我们设置了一个场景，即每个被评价的啤酒风格的平均评价。接着，我们将被评价过的 beer_style_id 设为 1。

最后，我们进入实际的回归问题，正是在这里我们尝试着将综合评价映射到啤酒风格。从这里开始，我们对收缩参数进行迭代，直到矩阵可逆，以便我们进行回归计算。最终，我们找到斜率参数并将其返回。

你可能想知道我们用这个斜率参数可以做什么。它只是一个关于某人对某种啤酒喜好程度的斜率。我们可将其存入表格 user_preferences 来对用户偏好持久化。该表格中包含了 beer_style_id 和一个偏好值。从该表中，我们可图形化地从最顶端的偏好移动到其他也对同一品牌的啤酒进行评价并且也喜欢它的评论者。

这是一种形式的协同过滤。我们识别了用户的偏好。利用它，我们可在图中移动到下一个用户。

9.6.5 利用用户偏好实现协同过滤

为了实现某种形式的协同过滤，我们希望找到具有相似口味的用户，然后找到他们所喜爱的、我们尚未尝试过的商品。为此，我们需要编写下列代码：

```
# lib/models/reviewer.rb

class Reviewer < Sequel::Model
  def friend
    skip_these = styles_tasted - [favorite.id]
```

```
    someone_else = UserPreference.where(
      'beer_style_id = ? AND beer_style_id NOT IN ? AND reviewer_id != ?',
      favorite.id,
      skip_these,
      self.id
    ).order(:preference).last.reviewer
  end

  def styles_tasted
    reviews.map { |r| r.beer.beer_style_id }.uniq
  end

  def recommend_new_style
    UserPreference.where(
      'beer_style_id NOT IN ? AND reviewer_id = ?',
      styles_tasted,
      friend.id
    ).order(:preference).last.beer_style
  end
end
```

这些代码会生成一个方法 recommand_new_style，它会告知我们一种要品尝的新风格。最精华的部分在于我们不需要显式地进行测试，因为我们已经对偏好进行了测试。这便是我们所需要的全部。

9.7　小结

核岭回归方法是一种用于快速找到病态问题（即变量 x 的维数高于观测量的数目）的解的有用工具。这种问题经常在评论中出现，用户会对评价感到厌烦而离开。虽然你也可以使用其他方法，但岭回归绝对是一种性能极佳的工具。

如你所见，岭回归非常强大：我们找到了对啤酒风格的偏好，并实现了一种协同过滤算法。在计算完偏好后，该算法可找到相似的偏好，并据此向我们做出推荐。

第10章

模型改进与数据提取

有时，无论一个算法本身有多好，都无法奏效。有时情况甚至更糟，用它完全无法得到任何有意义的结果。数据中可能含有大量噪声，有时要找到问题的来源几乎是不可能的。本章将介绍能够提升已有算法性能的方法，包括选择更好的特征，以及将特征变换为新的特征集。为此，我们对与交叉验证或产品监测有关的度量指标进行了监测。

在模型改进策略方面，本章有些“自助餐”的味道。这是因为修正模型有很多方法。

10.1 维数灾难问题

前面介绍过，维数灾难对于基于距离的机器学习算法是一个很大的问题。一般而言，当维数升高时，数据之间的平均距离也会增大。我们以图 10-1 为例，可以看到该图中的球心位于坐标原点 (0,0,0)。

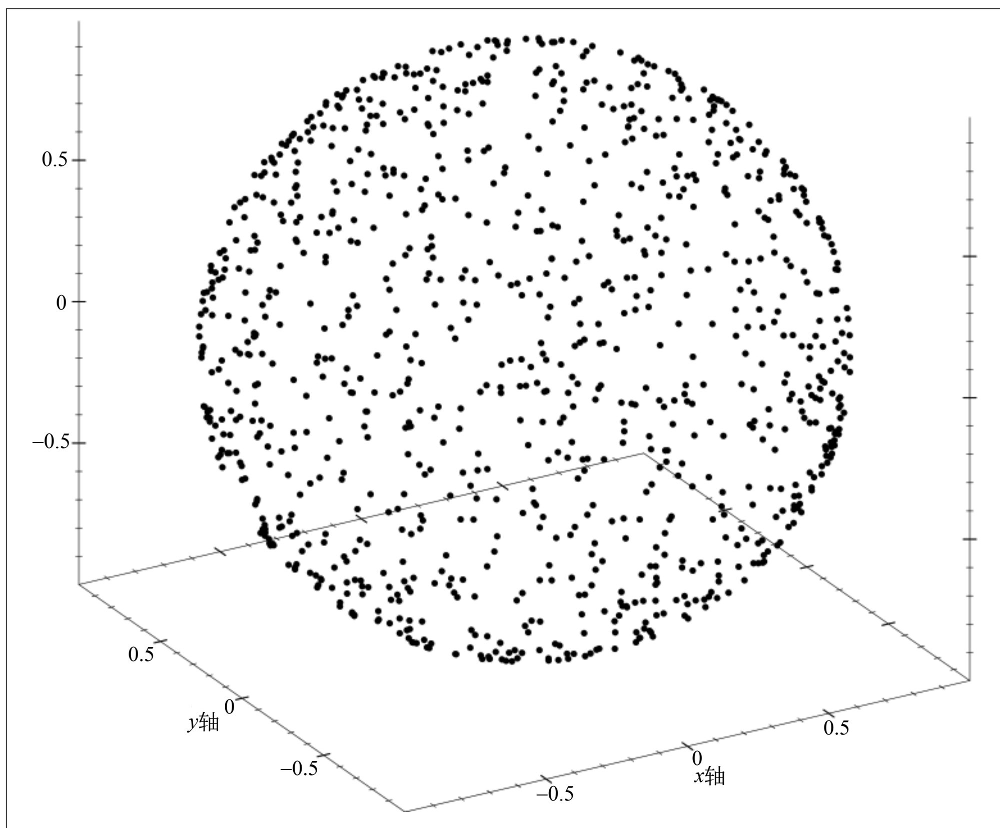


图 10-1：对于三维问题，数据点的平均距离为 1，因为数据都位于一个完美的单位球面上

这些点位于三维空间时，各点的平均距离为 1，但如果将这些数据投影到二维空间，情况会有什么变化？结果非常富有启发性（参见图 10-2）。

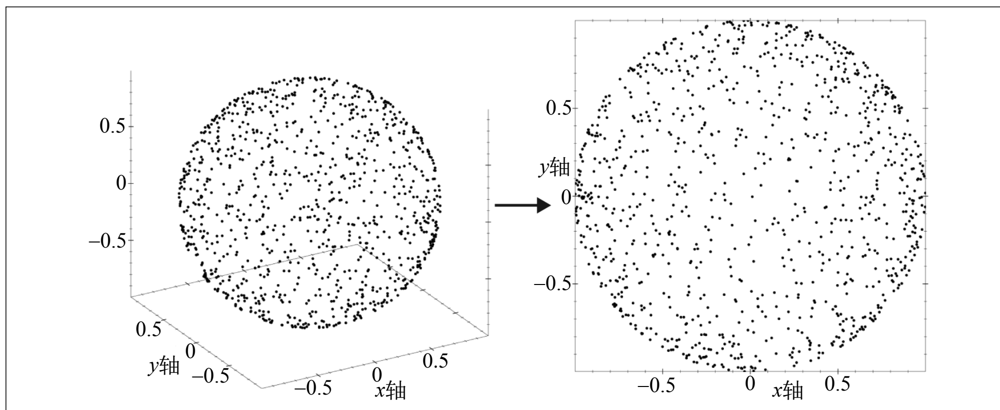


图 10-2：对于二维情形，数据点的平均距离变为 0.74

在上图中，左侧是一个单位球。这个单位球是用位于其外轮廓上的一些随机点创建的。右侧是一个圆，但它实际上是左侧的单位球投影到二维空间的结果。由于左图是一个单位球，因此各数据点之间的平均距离为 1，而在右图的圆中，各数据点的平均距离则变为 0.74。这意味着投影会使数据的维数降低，而数据点之间的平均距离会变小。在第 3 章中，我们通过在特征提取环节引入 SURF 算子来解决这个问题。我们并不试图找到所有像素的最近邻，而是需要使用一个较小的数据集。据我们所知，避免维数灾难的唯一途径是降维。

下面将讨论两种用于克服维数灾难的方法：特征选择和特征变换。

10.2 特征选择

我们不妨先来考虑一些没有实际意义的数据。比方说，我们希望测量天气数据，并希望能够在给定三个变量（咖啡消耗量、冰激凌消耗量和季节）的条件下预测温度。详情请参阅表 10-1。

表10-1：与Matt的冰激凌和咖啡消耗量相关的西雅图天气数据

平均温度	Matt的咖啡消耗量	Matt的冰激凌消耗量	月份
47F	4	2	1 月
50F	4	2	2 月
54F	4	3	3 月
58F	4	3	4 月
65F	4	3	5 月
70F	4	3	6 月
76F	4	4	7 月
76F	4	4	8 月
71F	4	4	9 月
60F	4	3	10 月
51F	4	2	11 月
46F	4	2	12 月

从该表可以看出，我每天大约要喝四杯咖啡。我在夏季会吃更多的冰激凌，那时气温通常偏高一些（如图 10-3 所示）。

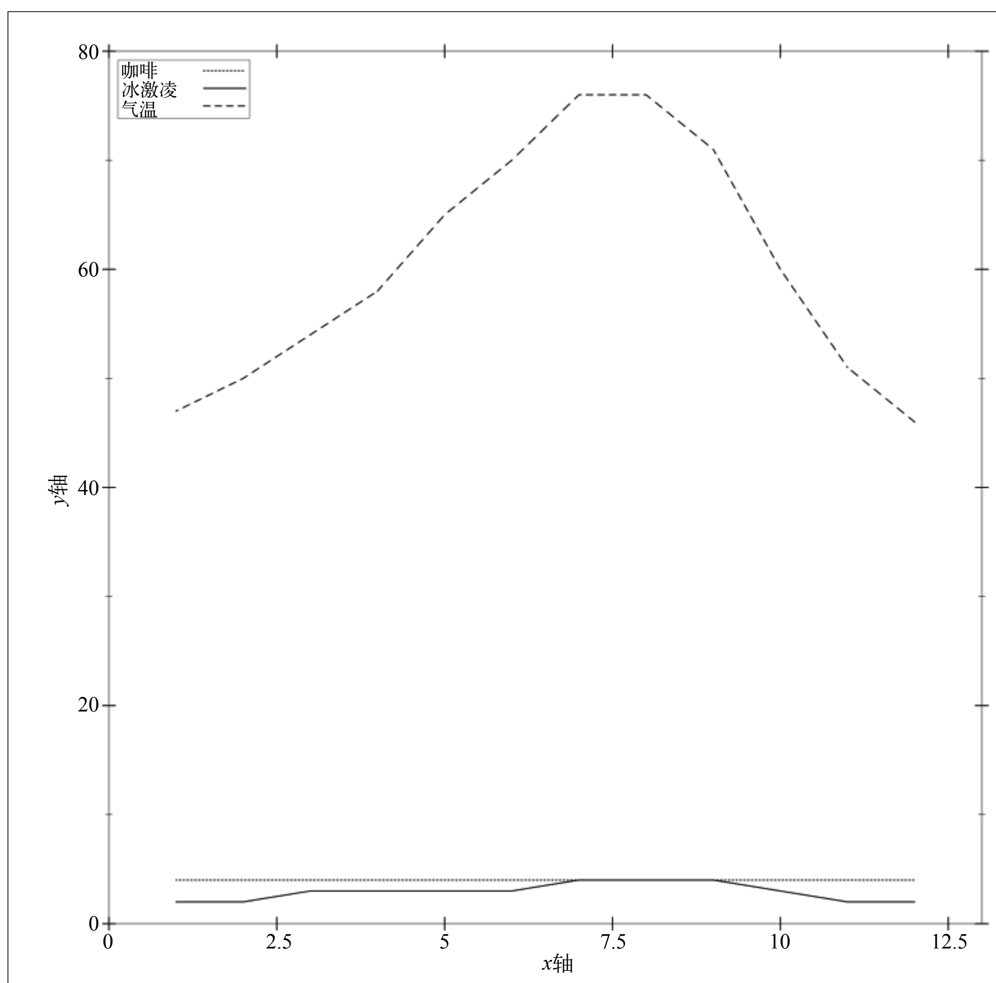


图 10-3：我一年的冰激凌消耗量情况

我们知道，季节是引起气温变化的原因之一。通常夏季温度较高，而冬季温度较低。这是因为太阳处在天空中位置的不同。冰激凌消耗量并不重要，咖啡消耗量也不重要，但冰激凌消耗量与较热的月份的确存在某种相关性。

但这里要意识到的是，我们有一个不相关特征，它会使数据向它偏斜；我们有一个组成变量，它实际上是模型外其他变量的组合；最后我们有一个信号，但它是月份。

例如，我们希望随机选取一个变量子集，并测试是否有性能提升。我们的确可以这样做。这种做法称为随机特征选择，它通常需要花费大量时间。其原因在于数据在较低维的空间中具有良好的行为。因此，即便我们采用随机降维的方式，数据本身也得到了改进。

这种降维方法的基本思想是随机选取一个数据子集，然后将模型运用于该子集。随机特征选择可能是最简单的模型改进方法之一。

但这种做法有一个很大的问题，即效率太低。不幸的是，将特征选择包裹在模型中需要花费大量时间。这是因为你需要随机抽取数据，选择某个子集，接着运行你的模型，然后在测试其拟合效果。正如你所预计的，这个过程需要花费很多时间。不过，对于 K 近邻算法和其他快速算法，这种策略可能已经足够好了。但如果你准备为一个神经网络模型提供更好的数据或可能需要花费一些时间的东西，该如何应对？在这种情况下，我们可以对数据进行过滤，即便是在数据送入算法之前。

10.3 特征变换

为理解特征变换，我们以记录食物摄取量为例。很多人都会追踪饮食中的卡路里情况。假设你希望追踪自己何时有饥饿感，何时没有饥饿感。因此，你通过一份日志来记录自己一天中哪些时段有饥饿感，以及饥饿的程度。

唯一的问题是，在记录饥饿情况期间，你恰好在洛杉矶和夏威夷两地之间往来。你收集的数据如表 10-2 所示。

表10-2：饥饿日志

是否感到饥饿	时间	时区偏移
是	7	–8
否	8	–10
否	9	–8
是	9	–10
是	12	–8
否	14	–10
否	15	–8
否	16	–8
否	18	–10
否	19	–10
是	18	–8
是	20	–10

这组数据中含有大量噪声，何时产生饥饿感非常不明确，如图 10-4 所示。

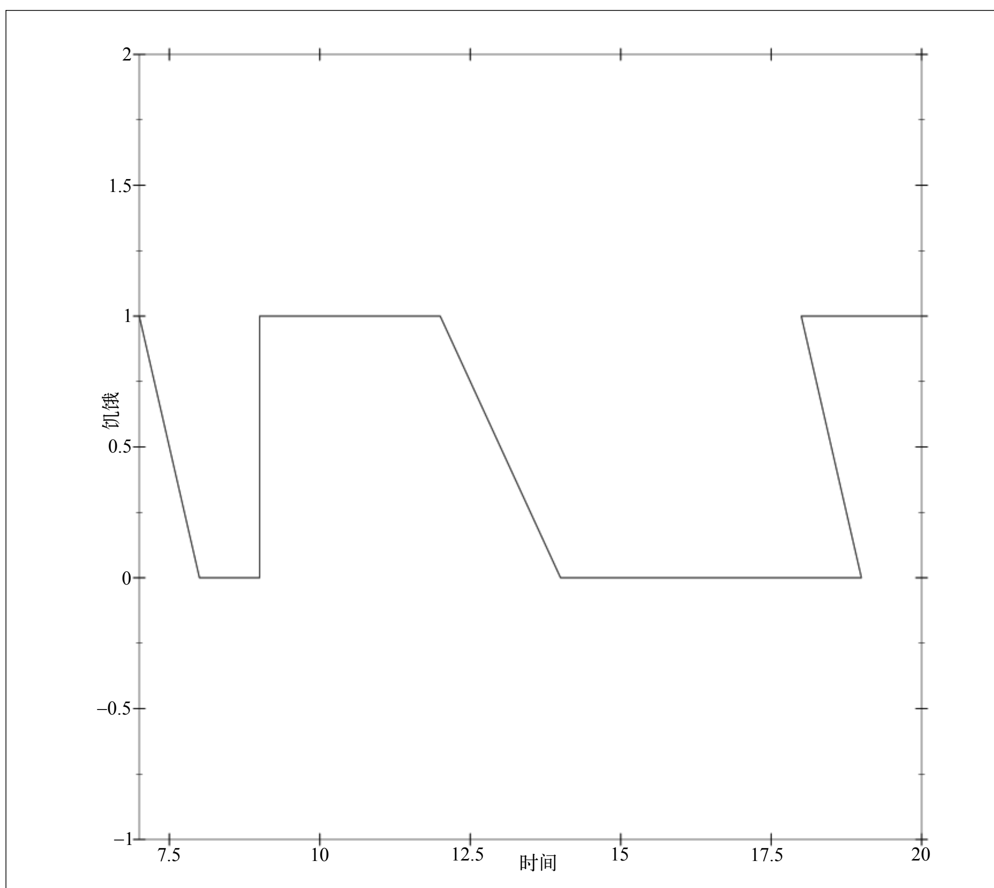


图 10-4：该图所呈现的趋势非常不明确

可以看出，你的饮食似乎不存在任何模式。你只是有时感到饥饿，有时不饿。

但如果将时间和时区偏移通过线性组合整合在一起，你会注意到你每天会在三个时段感到饥饿，即 7 点、12 点和晚 6 点。由此可得到一幅与上图完全不同的图（如图 10-5 所示）。

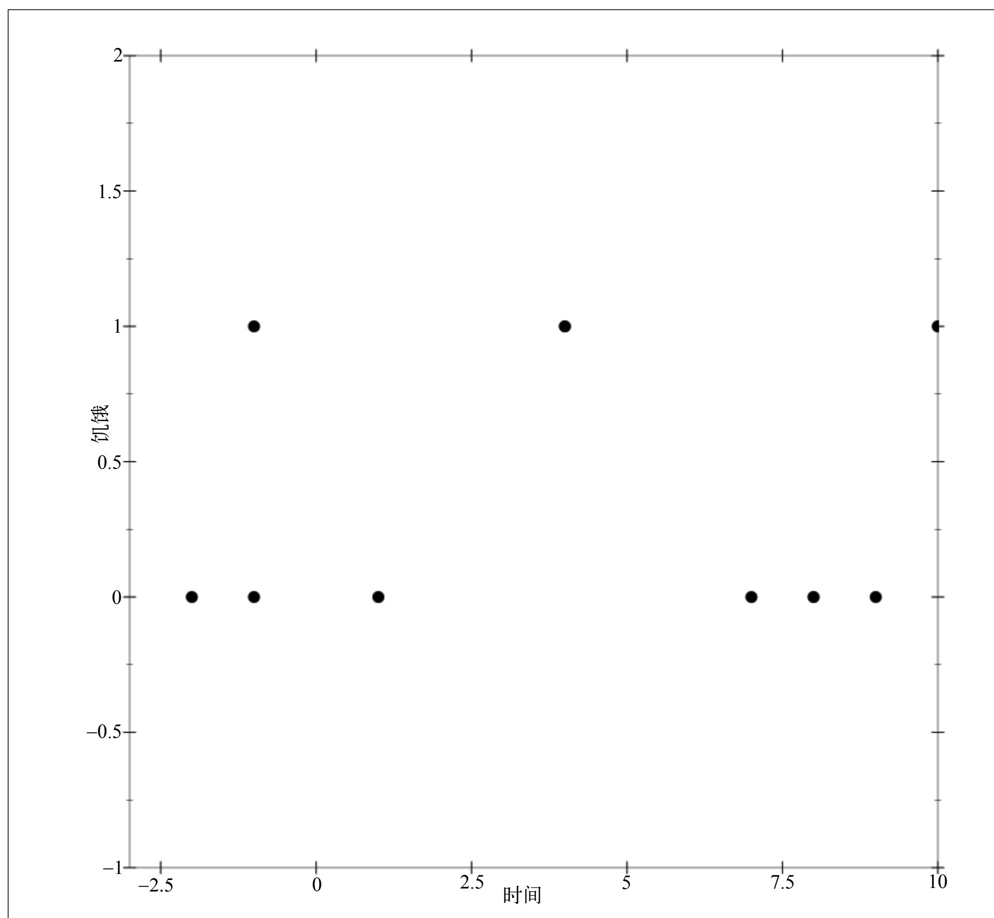


图 10-5：特征变换为时区偏移和时间之和

实际上，还有一种更好的方式，即借助*特征变换算法*。特征变换算法有多种类型，本章中我们将只关注 PCA 和 ICA。

10.4 主分量分析

主分量分析（Principle Component Analysis，PCA）是一种存在已久的方法。这种算法只关心方差最大的方向，并将其定为第一主分量。这与回归的原理非常类似，因为后者的核心也是确定数据映射的最佳方向。假设你拥有一个含噪声的数据集，如图 10-6 所示。

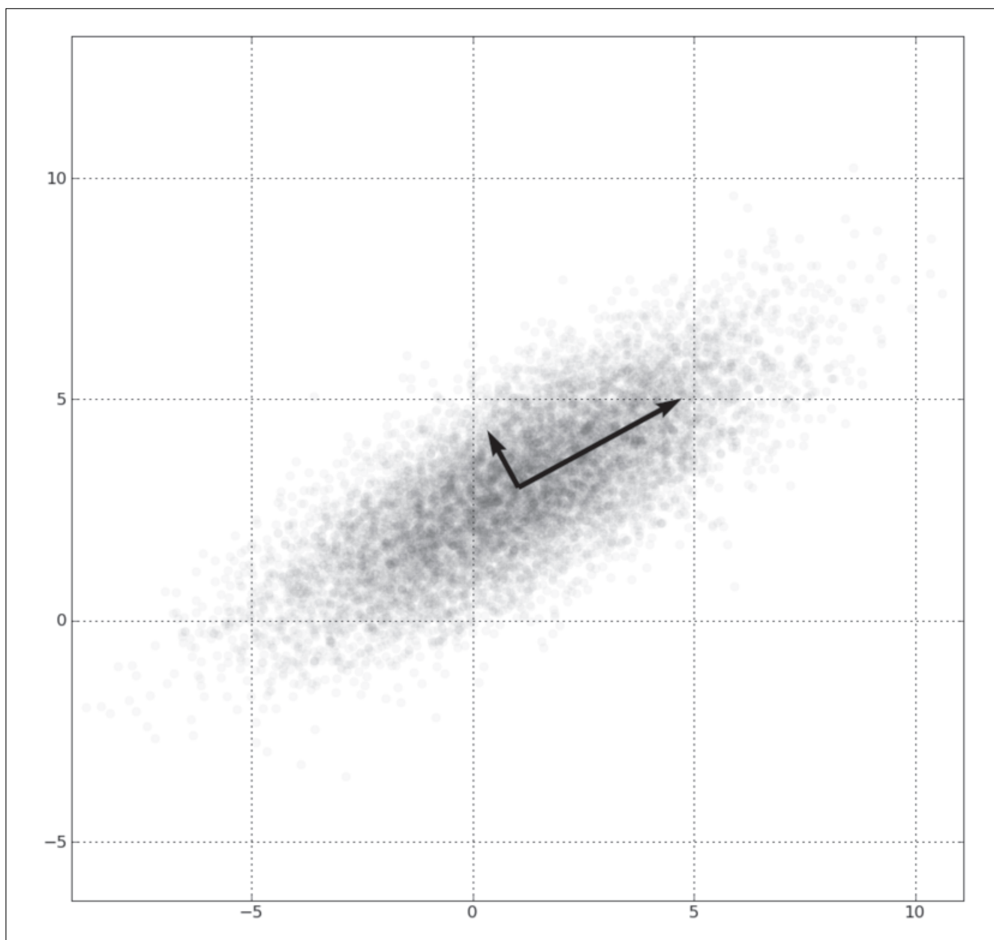


图 10-6：散点图中的主分量

可以看到，这个数据集具有明确的方向，即右上方。如果要确定主分量，显然应当是右上方，因为数据在这个方向上具有最大的方差。第二主分量与第一主分量正交，通过迭代，便可将数据集的维度变换到这些主分量方向上。

另一种理解 PCA 的方式是考虑它与人脸图像的关系。当你将 PCA 运用于一组人脸图像时，会出现一种被称为 Eigenfaces 的有趣结果（如图 10-7 所示）。

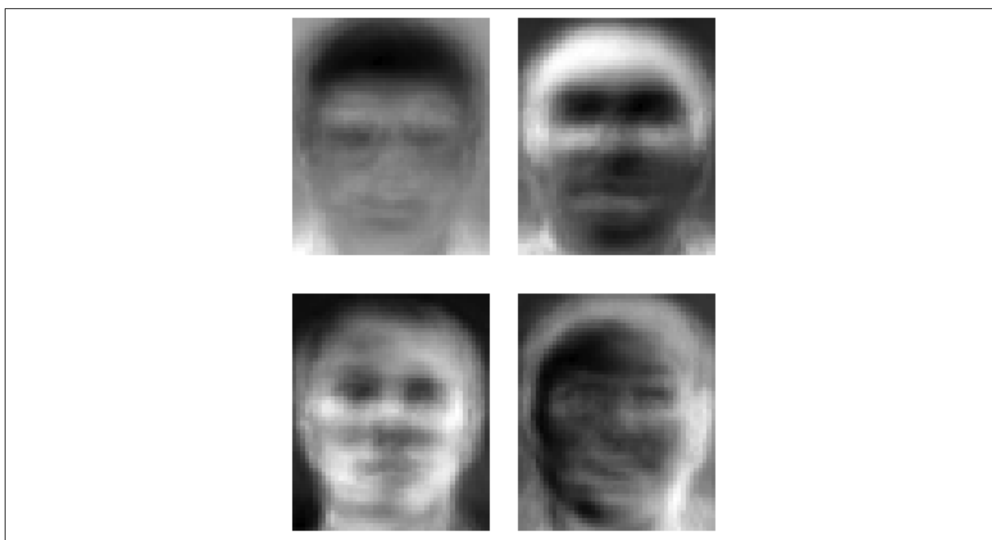


图 10-7：由剑桥大学的 AT&T 人脸库得到的 Eigenfaces

虽然这些图像看起来有些怪异，但有趣的是它们实际上是对所有训练数据取平均而得到的平均脸。现在，我们暂时不去实现 PCA，而是要待到下一节实现 ICA 时一起实现，因为后者实际上是依赖 PCA 的。

10.5　独立分量分析

假设你正参加一个聚会，你的朋友走过来与你交谈。你的旁边有个你很厌烦的人在喋喋不休，而房间的另一端有一台持续发出噪音的洗衣机（参见图 10-8）。

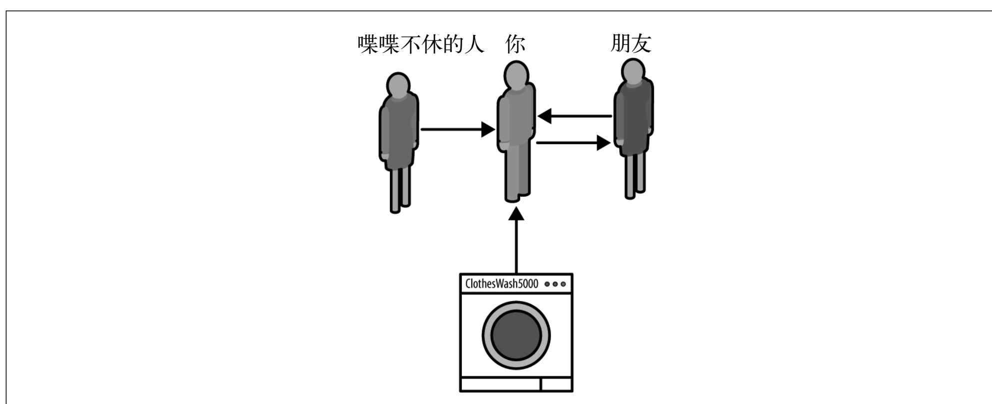


图 10-8：鸡尾酒会示例

你希望听清楚朋友的话，因此会选择近距离倾听。作为人类，我们很容易区分洗衣机的噪

声和其他人的喧哗声。但对于数据，我们如何能够做到这一点？

比方说，你不是在倾听朋友的谈话，而是有一段录音，你希望将背景中的噪音滤除。那么我们应当如何实现这个目标？你可使用一种称为独立分量分析（Independent Component Analysis，ICA）的算法。

从技术上讲，ICA 最小化的是互信息，或两个变量之间共享的信息。直观上看，这是有意义的：从混合信号中找到存在差异的信号。

与图 10-7 所示的人脸识别示例相比，ICA 提取的是什么样的特征？与 Eigenfaces 不同，ICA 提取的是一些面部特征，如鼻、眼和头发等。

这两种算法对于数据变换都非常有用，并可对信息做进一步分析（参见图 10-9）。

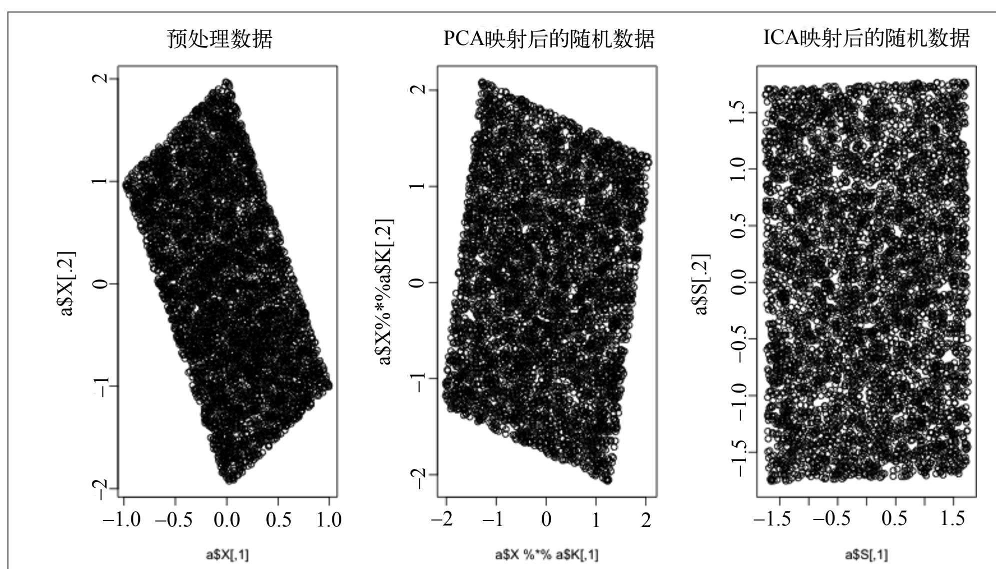


图 10-9：利用 PCA 和 ICA 映射后的随机数据

不幸的是，与 PCA 一样，也不存在 ICA 的 Ruby gem 包。不过，我们可以利用“R in Ruby”gem 来调用 R。对于一本基于 Ruby 语言的机器学习书籍而言，这绝对是一种“欺诈”行为，但有时调用其他语言也是明智之举。当然，还有另一种方案，即使用 JRuby 和 FastICA。实际上，这也正是 R 语言所采用的。



安装说明

要运行本章示例，请在 Github 上获取本书的配套代码：https://github.com/thoughtfulml/examples/tree/master/9-improving-models-and-data-extraction。

为正常运行示例，你还需要安装 R 软件。

我们所要做的第一步是在 R 中安装 FastICA。为此可键入下列语句：

```
# example_1.rb

require 'rinruby'

R.eval(<<- R)
  install. packages("fastICA" )
  library(fastICA)
  S <- matrix(runif(10000), 5000, 2)
  A <- matrix(c(1, 1, - 1, 3), 2, 2, byrow = TRUE)
  X <- S %*% A
  a <- fastICA(X, 2, alg.typ = "parallel", fun = "logcosh", alpha = 1,
  method = "C", row.norm = FALSE, maxit = 200,
  tol = 0.0001, verbose = TRUE)
  par(mfrow = c(1, 3))
  plot(a$X, main = "Pre-processed data")
  plot(a$X %*% a$K, main = "PCA components" )
  plot(a$S, main = "ICA components" )
R
```

至此，我们已经了解了特征变换、特征选择以及其他相关内容。下面我们进入本章的最后一部分——在产品环境中监测机器学习算法的性能。

10.6　监测机器学习算法

本书介绍了测试编写、交叉验证和奥卡姆剃刀准则，但极少涉及代码的监测。如果你觉得产品代码看起来已经足够好了，这只能说明度量的力度不够。对于机器学习，情形有些差异：我们不能因为这些算法看起来很奇妙就在部署之后停止代码测试。

我们可利用的工具通常度量的是精度和查全率，并且在异常出现时向我们发出警告。稍后我们还将介绍一种度量均方误差的在线方法。

10.6.1　精度与查全率：垃圾邮件过滤

你是否还记得我们的垃圾邮件过滤器？简言之，我们希望将每封邮件标记为垃圾邮件或普通邮件。由于电子邮件非常重要，所以我们宁愿精度低一些，即让某些垃圾邮件进入收件箱，而不至于因误分类导致重要的信息丢失。

假设我们认为表 10-3 所示的邮件数据符合实际情况。

表10-3：依据经验得到的垃圾邮件和普通邮件分类结果

	预测为普通邮件	预测为垃圾邮件
普通邮件	10 000	100
垃圾邮件	40	60

可以看出，我们将 10 000 封普通邮件正确地分类为普通邮件。不幸的是，有 100 封普通邮件被误分类为垃圾邮件。

这种信息十分有价值，在进行交叉验证时，可利用它来优化我们的模型。但如果我们希望进行监测，应如何做？这正是精度（precision）、准确率（accuracy）和查全率（recall）的用武之地（如图 10-10 所示）。

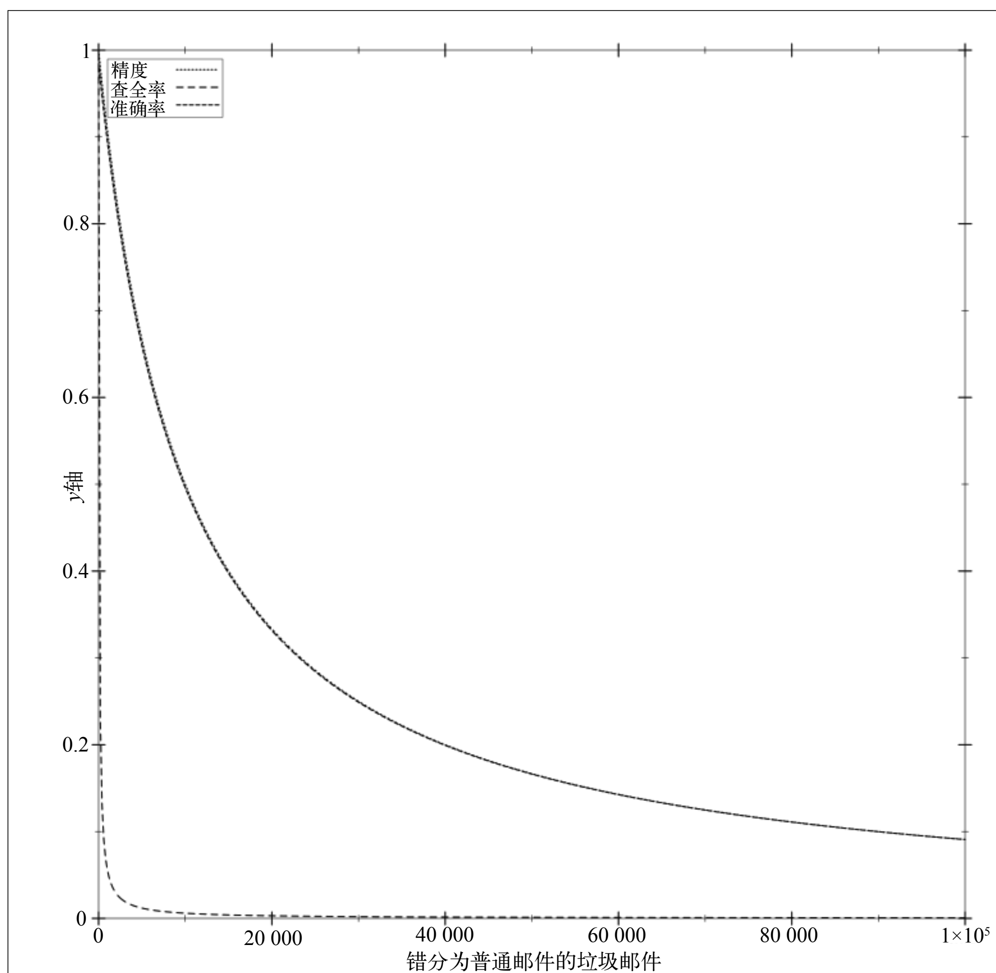


图 10-10：精度、查全率和准确率的衰减

*精度*是被正确分类的垃圾邮件数目与被预测为垃圾邮件的数目之比。通过上表可以看出，精度为 60/160=3/8。这个指标的确不高。这意味着该算法会给我们提供很多与垃圾邮件不相关的样例。因此，它会错误地认为一些普通邮件是垃圾邮件。

*准确率*是被正确分类的样本数除以样本总数。因此，在上例中，准确率为 (10 000+60)/

(10 000+100+40+60) ≈ 98.6%。这个结果看起来不错，这意味着有 98.6% 的样本的分类结果都是正确的。本书大量使用了这个比指标来度量错误率。

最后，上例的查全率是 60/(60+40)=3/5。这意味着我们可依据分类结果得到一半以上的垃圾邮件，这个结果是令人满意的。

这些度量对于交叉验证是有价值的，但我发现它们更适合用于监测。当用户有交互行为，并将更多数据输入该算法时，记录所有这些指标便非常重要。

例如，假设现有垃圾邮件过滤器的过滤效果不够令人满意，而人们不断地将新的普通邮件标记为垃圾邮件。这会对我们的度量产生什么影响？答案可从表 10-4 中找到。

表10-4：数据支持图

被错误地预测为普通邮件	普通邮件的精度	查全率	准确率
0	100%	100%	99%
10	99%	85%	98.9%
20	99%	75%	98%
30	99.7%	66%	98%
…	…	…	…
60	99.4%	50%	98%
…	…	…	…
100	99%	37.5%	98%

可以看出，就准确率而言，假负例对于包含大量训练样本的问题影响很小，但对查全率则不然。当查全率跌至 50% 以下时，模型将不可用，应当产生监测警示。

10.6.2　混淆矩阵

我们已经讨论过了假的和正的预测，但对此其实还有一个一般性的术语——混淆矩阵（confusion matrix）。该术语与被正确分类的实例有关。因此，例如我们需要将啤酒划分为三个类别：Pilsner、Stout 和 Hefeweizen。给定一个分类算法，我们发现分类结果如表 10-5 所示。

表10-5：混淆矩阵

	Pilsner	Stout	Hefeweizen
Pilsner	20	1	3
Stout	1	30	1
Hefeweizen	5	1	10

由于酒的种类不尽相同，因此我们的算法找到了一个适合于给定样本集的分类策略。依据该混淆矩阵，我们可计算查全率、精度和模型的准确率。

例如，Stout 类的分类精度为 30/32=93.75%，而该类的查全率也为 30/32=93.75%。

不幸的是，混淆矩阵有一个缺陷——它们仅适用于离散分类问题。对于像回归或返回一个连续变量的算法，应当如何度量误差？为此，我们可计算均方误差。

10.7 均方误差

当需要评价某个预测结果为连续值的算法时，需要采取与上述不同的方法，即度量模型的均方误差。但在产品环境中，要计算该指标是有一定难度的，因为我们需要保存过去所有的分类结果和误差。实际上，我们完全可以使用其他策略来规避这一问题。

例如，我们的目标是计算当新的信息到来时系统随时间变化的均方误差。比方说，有如下模型：

$$\hat{y} = f(x)$$

其中，y 为实数。假设我们可从用户那里获取一些信息，如他们给出的评分，因此我们可计算误差：

$$\epsilon = (\hat{y} - y)^2$$

这里对误差取平方的目的是使误差为正。

在很多人来看，我们应当保存所有的误差，并按下式计算总误差：

$$\sum_{i=0}^{n} \epsilon_i$$

但我们完全可以以增量方式计算均方误差。我们首先给出平均误差的表达式：

$$\overline{\epsilon_n} = \frac{\epsilon_1 + \epsilon_2 + \cdots + \epsilon_n}{n}$$

同样，接下来的一轮均值可表示为：

$$\overline{\epsilon_{n+1}} = \frac{\epsilon_1 + \epsilon_2 + \cdots + \epsilon_n + \epsilon_{n+1}}{n+1}$$

现在，如果将第一个式子乘以 n，可将第二个式子改写为：

$$\overline{\epsilon_{n+1}} = \frac{n * \overline{\epsilon_n} + \epsilon_{n+1}}{n+1}$$

这意味着下一个均值等于前一个均值乘以所使用的实例数目，加上当前误差，并除以到目

前为止的实例总数。

因此，比方说我们通过 10 次迭代得到当前均方误差为 2。我们在第 11 次迭代时发现平方误差为 100。这意味着新的均方误差为 (2*10+100)/11 ≈ 10.9。

这意味着我们很容易编写用于在产品代码中监测均方误差的程序。

```
# incremental_meaner.rb

class IncrementalMeaner
  attr_reader :current_mean, :n
  def initialize
    @current_mean = 0
    @n = 0
    @mutex = Mutex. new
  end

  def add(error)
    @mutex. synchronize {
      @current_mean = ((@n * @current_mean) + error) / (@n + 1. 0)
      @n += 1
      @current_mean
    }
  end
end
```

10.8　产品环境的复杂性

正如有些人所言，任何会出错的事情都有可能出错。对于产品环境尤其如此。我们可进行交叉验证，测试我们的接口，并确定我们的模型是否表现良好，但对于产品环境而言，仍存在崩溃的可能。用户输入我们无从优化，因为最有趣的事情都是人做出的。因此，对均方误差以及精度、查全率等度量构建监测程序，对于度量算法性能非常重要。

要进行监测，还可利用一种体系组件，即在算法中需要引入反馈环。换言之，我们需要对某些内容进行测试。否则，代码将无法正常工作，而最终网站中的信息缺乏变化，如一潭死水，从而变得毫无用处。最后，用户体验才是我们试图借助机器学习算法加以优化的。

10.9　小结

本章介绍了多种改进已有模型的方法。有时，可以只选择那些表现更好的特征，有事则需要对已有特征进行变换。但最重要的是确保我们依据某个基准对结果进行度量，并在产品环境中或通过用户监测模型是否正常。

第11章

结语

下面我们进入本书的最后一章。到目前为止，你对机器学习的理解很可能仍然不及机器学习专业的博士那样深刻，但我希望你有所收获。我希望，对于机器学习所擅长解决的那些问题，你已培养了一种思维过程。我坚信，测试是有效使用这种科学方法的唯一途径。这正是现代社会之所以存在的原因，测试还有助于我们编写高质量的代码。

当然，要想对一切问题编写测试是不大可能的，但保持这种习惯至关重要。我希望你已经掌握了一些将这种习惯运用于机器学习问题的方法。本章将从更高的层次来讨论之前介绍过的各种方法，我还将向你推荐一些阅读材料，以便你更深入地开展机器学习研究。

11.1 机器学习算法回顾

我们之前曾经介绍过，机器学习方法可分为三大类：有监督学习、无监督学习和强化学习（参见表 11-1）。本书并未涵盖强化学习，但鉴于你目前已经具备了良好的基础，我强烈建议你对其进行研究。在本章的最后，我会向你推荐一些有关强化学习的阅读材料。

表11-1：机器学习方法的类别

类别	描述
有监督学习	有监督学习是机器学习中最常见的类型。它本质上是一种函数逼近。我们试图将数据点映射为一个模糊函数。通过优化，我们希望依据训练数据拟合出一个与未来数据取得最佳逼近效果的函数。该类方法之所以称为“有监督方法”，是因为它们需要接收一个训练集或学习集
无监督学习	无监督学习只分析数据，而不向某个 Y 映射。该类方法之所以称为“无监督方法”，是因为它们并不知道输出结果应为何物，而是需要自己提供
强化学习	强化学习与有监督学习相似，但会对每一步生成一个“回报”。例如，好比一只在迷宫中寻找奶酪的老鼠，它希望找到奶酪，但绝大多数时候它不会得到任何奖励，除非最终找到奶酪

对于上述每类方法，一般都有两类偏差。一种是约束偏置（restriction bias），另一种是偏好（preference）。约束偏置限制的是算法本身，而偏好则反映了算法适用于解决何种问题。

所有这些信息（如表 11-2 所示）都有助于我们确定是否应当选择某种算法。

表11-2：机器学习算法矩阵

算法	类型	类别	约束偏置	偏好偏置
KNN	有监督学习	基于实例的	一般说来，KNN 适合度量基于距离的逼近；易受维数灾难的影响	适于求解基于距离的问题
朴素贝叶斯	有监督学习	概率的	适用于那些输入相互独立的问题	适用于那些各类概率值为正的问题
SVM	有监督学习	决策面	适用于两类分类中具有明确界限的问题	适用于两类分类问题
神经网络	有监督学习	非线性函数逼近	几乎没有约束偏置	适合二元输入问题
（核）岭回归	有监督学习	回归	对所能解决的问题具有很低的约束偏置	适用合于连续变量
隐马尔科夫模型	有监督 / 无监督	无后效性	适用于那些符合马尔科夫假设的系统信息	适用于时间序列数据和无记忆的信息
聚类	无监督	聚类	无限制	适用于给定某种形式的距离（欧氏距离、马氏距离或其他距离）时，数据本身具有分组形式
过滤	无监督	特征变换	无限制	适用于数据中有大量变量需要过滤的场合

11.2　如何利用这些信息来求解问题

利用表 11-2 所示的算法矩阵，我们可明确如何解决一个给定问题。例如，对于确定某人居住的社区这样的问题，KNN 便是一个很好的选择，而朴素贝叶斯分类模型则丝毫派不上用场。但朴素贝叶斯分类模型可以确定情绪或其他类型的概率。对于寻求两类数据划分边界的问题，支持向量机算法则非常适合，而且不易受维数灾难的影响。因此，对于拥有大量特征的文本问题，支持向量机通常都是很好的选择。神经网络可以求解从分类到自动驾驶这样范围很广的问题。核岭回归则是向线性回归模型中添加了一种简单的技巧，并且能够找到曲线的均值。隐马尔可夫模型能够追踪乐谱，标注词性，并适用于其他类似于系统的应用。

聚类算法适合于那些不含明确输出的数据分组问题。这类算法对于数据分析非常有帮助，也可用于构建数据库或高效地保存数据。过滤方法非常适用于克服维数灾难。在第 3 章中，为将所提取到的像素转换为特征，我们曾大量使用了该类方法。

本书尚未谈及的一点是，学习这些算法仅仅是一个开始。最重要的是，我们应当认识到，选择什么方法并不是最关键的，要尝试解决的问题才是最重要的。这正是我们使用交叉验证、度量精度、查全率和准确率的原因。对每一个步骤进行检查和测试，保证了我们至少在接近更优的答案。

我建议你了解更多的机器学习模型，并思考如何将测试应用于这些方法。大多数算法都内置了相关的测试，但为了编写高质量的可随时间学习的代码，作为人类，我们也需要对自己的工作进行检查。

11.3 未来的学习路线

至此，我们的学习之旅才刚刚开始。机器学习是一个快速发展的领域。我们可以学习如何利用深度学习网络来实现自动驾驶，以及如何利用受限玻尔兹曼机来对像健康问题这样的问题进行分类。机器学习的未来无比光明。通过阅读本书，你已经奠定了良好的基础，可以研究更深入的子话题，如强化学习、深度学习、人工智能以及更复杂的机器学习算法。

有太多的信息和资料等待你去研究。下面给出一些推荐阅读的资源：

- Peter Flach, *Machine Learning: The Art and Science of Algorithms That Make Sense of Data* (Cambridge, UK: Cambridge University Press, 2012).
- David J. C. MacKay, *Information Theory, Inference, and Learning Algorithms* (Cambridge, UK: Cambridge University Press, 2003).
- Tom Mitchell, *Machine Learning* (New York: McGraw-Hill, 1997).
- Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd Edition (London: Pearson Education, 2009).
- Toby Segaran, *Programming Collective Intelligence: Building Smart Web 2.0 Applications* (Sebastopol, CA: O’Reilly Media, 2007).
- Richard Sutton and Andrew Barto, *Reinforcement Learning: An Introduction* (Cambridge, MA: MIT Press, 1998).

此外，你还可观看在线课程或 YouTube 上的大量视频资料。学习与深度学习有关的讲义是非常有益的。Geoffrey Hinton 的讲义（http://videolectures.net/geoffrey_e_hinton/）是绝佳的选择，你也可参考 Andrew Ng 的讲义，包括他的 Coursera 课程（https://www.coursera.org/instructor/~35）。

既然对机器学习已经有所了解，你可以尝试着去解决一些不是非黑即白，而是涉及很多“灰色”的问题。借助本书自始至终所贯穿的测试驱动的方法，你在观察和思考这些问题时，就像配备了科学的眼镜，而且不再局限于辨别真伪，而是在准确性上前进了一大步。机器学习是一门充满魅力的学科，因为它使得两个独立的思想——具有坚实理论基础的计算机科学和含有噪声的数据——整合在一起，形成了无比美妙又和谐的关系。

作者介绍

Matt Kirk 是一名程序员，但他不在湾区生活。虽然从事程序设计工作已 15 年有余，但无论对什么事情他都把自己当成初学者。他对学习和构建工具的热爱是其职业生涯的不竭动力，这些工具涉及的领域极为广泛，包括金融、创业、钻石、重型机械和伐木。本书不止是机器学习的精粹，而且也浓缩了 Matt 广泛的求知欲和对学习的热爱。他在全球许多会议中都发表过演说，但仍然十分享受每日的编程工作。在编写软件之余，他很可能去学习一些新事物，无论是园艺、音乐、木艺，还是改动刹车盘。

封面介绍

本书封面中的动物是一只欧亚雕鸮，正如其名称，这个物种主要生活在欧亚大陆。雌性雕鸮的翼展可达 74 英寸，体长可达 30 英寸，雄性雕鸮的体长和翼展略小。雕鸮是现存体型最大的鸮，其耳羽非常独特，眼睛为橙色。它们的下腹部非常柔软，且带有大量深色条纹。

雕鸮多出没于山地或松柏林中，为夜行猛禽，其猎物包括小型哺乳类动物、爬虫类、两栖动物、鱼类、体型较大的昆虫和蚯蚓。雕鸮喜欢在隐秘的地点繁衍后代，如山壑或岩石中。在巢穴中，它们间隔产卵可多达 6 枚，并在不同的时间进行孵化。产完这些卵后，雌性雕鸮会一心孵化，并哺育幼鸮，而雄性雕鸮则负责为雌性雕鸮和幼鸮觅食。待所有卵孵化完毕后，它们还会在接下来的五个月中亲代养育。

欧亚雕鸮可发出多种声音，包括鸣叫，其声音具有很强的穿透力。这是一种低沉的“呜呼”声。雄性雕鸮会强调第一个音节，而雌性雕鸮则会发出更高音调的“呃呼”鸣叫。当感受到威胁时，雕鸮会用类似点钞的声音或像猫那样吐东西来表达烦躁不安的情绪，有时也会采取一种防御性的姿势，如低头、将羽毛竖起、展开尾羽或张开双翼。

健康的成年雕鸮没有天敌，这使得它们位于食物链的最顶端，虽然它们有时也会遭到更小的鸟类（如鹰或其他鸮类）的围攻。然而，这个物种的头号死因却应归咎于人类：很多雕鸮都因触电、交通事故和射杀而亡。在野外环境中，雕鸮一般能存活 20 年左右；而人工饲养的雕鸮，由于无需面对恶劣的自然条件，寿命一般会更长。据报道，有些雕鸮在动物园这样的环境中会存活长达 60 年之久。欧亚雕鸮的栖息地横跨欧亚大陆 1200 万平方英里的土地，据估计，其种群数量在 250 000 和 250 万之间，是国际自然保护联盟“关心最少”的一个物种。通常在人迹罕至的地带，它们会大量聚集。不过，人们在欧洲的一些农场和类似公园的地方也发现有雕鸮出没。

封面图出自 *Braukhaus Lexicon*。

O'REILLY®

机器学习实践：测试驱动的开发方法

本书介绍在开发机器学习算法时如何运用测试驱动的方法，捕捉可能扰乱正常分析的错误。这本实践指南从测试驱动开发和机器学习的基本原理讲起，展示了如何将测试驱动开发运用于若干机器学习算法，包括朴素贝叶斯分类器和神经网络。

任何机器学习算法都有一些传统的测试方法，但它们通常都不会考虑编码中的人为错误。借助测试驱动的开发方法，你便不会像其他研究者那样盲目依赖机器学习的结果，而能够降低出错的风险，从而编写出整洁、稳定的机器学习代码。如果你熟悉Ruby 2.1，就已经做好了阅读本书的准备。

通过阅读本书，你将能够：

- **在编写代码之前，运用测试驱动的方法来编写和运行测试**
- **学习八种机器学习算法的最佳用法，并进行权衡**
- **通过动手实践真实示例，对每种算法进行测试**
- **理解测试驱动开发和对解进行验证的科学方法之间的相似性**
- **获悉机器学习的风险，如对数据产生欠拟合或过拟合**
- **探索可改善机器学习模型或数据提取的各种技术**

Matthew Kirk是Modulus 7公司的创始人，该公司针对数据科学和Ruby开发提供咨询服务。Matthew从事程序设计工作已15年有余，在全球许多技术大会上做过机器学习和数据科学主题的演讲。

“这本书非常有趣。对于想深入了解机器学习的开发者来说不可多得！”

——**Brad Ediger**

*Advanced Rails*一书的作者

“这本书真是太棒了！”

——**Starr Horne**

Honeybadger网站的联合创始人

“读完Matthew Kirk的《机器学习实践》，我收获颇丰。”

——**James Edward Gray II**

GraySoft公司的顾问

PROGRAMMING/MACHINE LEARNING

封面设计：**Ellie Volckhausen 张健**

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/人工智能

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-39618-1
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks